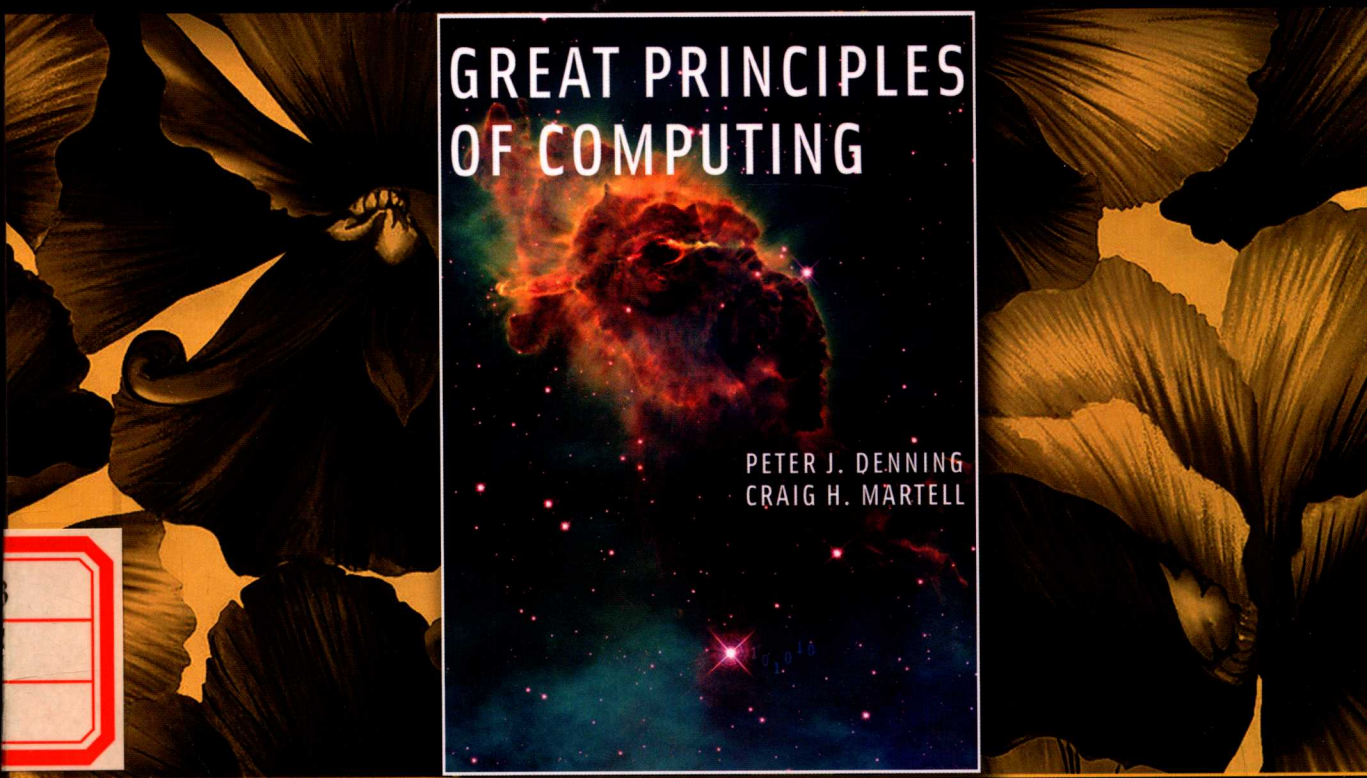
HZ BOOKS
华章教育

计 算 机 科 学 丛 书

伟大的计算原理

[美] 彼得 J. 丹宁（Peter J. Denning）
克雷格 H. 马特尔（Craig H. Martell） 著
罗英伟 高良才 张伟 熊瑞勤 译

Great Principles of Computing



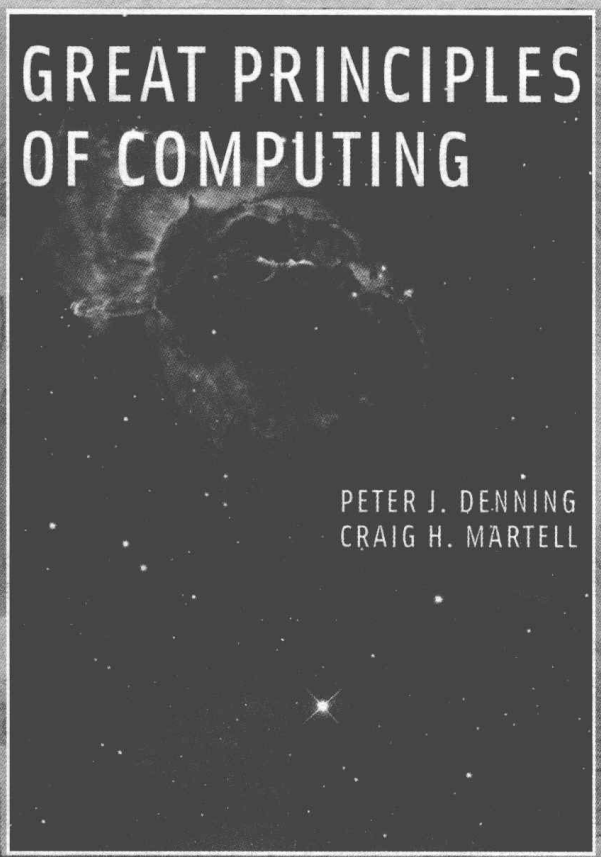
机械工业出版社
China Machine Press

机 科 学 丛 书

伟大的计算原理

[美] 彼得 J. 丹宁（Peter J. Denning）
克雷格 H. 马特尔（Craig H. Martell） 著
罗英伟 高良才 张伟 熊瑞勤 译

Great Principles of Computing



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

伟大的计算原理 /（美）彼得 J. 丹宁（Peter J. Denning），（美）克雷格 H. 马特尔（Craig H. Martell）著；罗英伟等译 . —北京：机械工业出版社，2017.4
（计算机科学丛书）
书名原文：Great Principles of Computing

ISBN 978-7-111-56726-4

I. 伟… II. ①彼… ②克… ③罗… III. 计算机科学 IV. TP3

中国版本图书馆 CIP 数据核字（2017）第 076369 号

本书版权登记号：图字：01-2016-3480

本书系统总结了从算法到系统横跨整个计算机领域的 6 类计算原理（计算、通信、协作、记忆、评估和设计），旨在构建起一个框架帮助读者认识计算思维，领会其核心思想——计算原理的相互影响以及问题有效解决的思维方式，并将计算思维运用到计算机科学以外的其他领域。

本书适合作为高等学校非计算机专业计算思维课程以及计算机专业计算机科学导论课程的教学参考书，也适合 IT 领域的程序员及专业人员阅读。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：和　静　　责任校对：殷　虹
印　　刷：北京市荣盛彩色印刷有限公司　　版　　次：2017 年 5 月第 1 版第 1 次印刷
开　　本：185mm × 260mm　1/16　　印　　张：16.25
书　　号：ISBN 978-7-111-56726-4　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum，Bjarne Stroustrup，Brian W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所

采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：（010）88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037

华章科技图书出版中心

计算机及其相关技术正在广泛而深刻地改变着我们的生活。没有哪门科学像计算机科学这样，能够与其他科学领域产生出如此多的交叉。尤其是近年来云计算、大数据、移动应用、机器学习、人工智能等，更是引起了人们的广泛关注。计算机是如此的神奇，但是人们却很少能够真正了解计算机是如何运行并完成复杂的工作的。人们对于计算机的理解似乎总是隔着一层面纱，有时清晰，有时却又很模糊，甚至很多计算机专业人员也同样会有这种感觉。

本书并不是单纯地介绍计算机技术，而是重新审视和理解计算本质，或者说阐述“计算之道”。本书从一个不同的视角，把计算看作是一门遵从一组基本原理的科学，所有计算机相关的技术都可以从这组基本原理中找到依托。具体而言，本书提供了一种重要原理框架（great principle framework）来刻画计算科学中最具基础性的基本原理。它将计算原理分为 6 类：通信、计算、协作、记忆（存储）、评估和设计，而在阐述时又将其所覆盖的范围分成 11 个更容易管理的模块，其内容涵盖了计算的方方面面——包括计算机是如何工作的、如何选择算法、计算系统是如何组织的，以及如何进行正确而可靠的设计等。

正如作者所说，这本书是为所有想利用计算科学及相关技术来实现特定目标的人而设计的。在介绍关于计算的基本原理时，作者深思熟虑地综合描述了计算背后的基本概念，并试图以一种“任何对计算有一定了解的人都能理解”的方式呈现。因此，与其他深入剖析特定技术的书籍不同，本书努力为读者构建一个关于计算的全面视图，而更深入的知识则需要（或者值得）读者自己不断去发掘、吸收，并将它们连贯起来。

本书的作者之一 Peter J. Denning 教授曾是 ACM 的主席，是若干基本计算原理的发现者，也一直非常关注计算科学的教育，并且早在 20 世纪 90 年代就开始本书内容的研究、整理以及教育实践。本书的作者发现，随着核心技术的增加，对于初学者来说，原理框架比技术框架更容易理解。本书的原理框架并不是重新定义计算的核心知识，但它确实提供了一种看待该领域并降低其表面复杂性的新方式。从这个意义上来说，基于原理的“计算”教学，可以帮助初学者抑或专业人员更加容易地构建一个关于计算的全面视图，是一种非常值得去实践的教育理念。另一方面，本书关于基本计算原理的讨论也一定会启发更多的人对“计算之道”进行广泛而深入的思考。

本书由北京大学计算概论课程组的几位老师共同翻译，具体分工如下：前言，第 8、9 章及相关内容由罗英伟、张彬彬翻译，并由罗英伟负责全书翻译工作的组织和协调；序，第 3、11、12 章及相关内容由高良才翻译；第 1、2、5、10 章及相关内容由张伟翻译；第 4、6、7 章及相关内容由熊瑞勤翻译。译稿之中倘若有不当之处，敬请读者批评指正。

译者

2017 年 3 月于北京大学

序

Great Principles of Computing

Peter J. Denning 和 Craig H. Martell 提出并研究了一个具有里程碑意义的话题：识别并阐明一些原理，使计算机做“我们希望的”并努力减少其与计算机“实际做的”（即我们的命令）之间的差异。bug（程序错误）就是这种差异的典型例子。bug 通常是由于编程中疏忽大意导致机器做了“我们所不希望的”而出现的一种结果。但是程序错误并不是 bug 的唯一来源和表现形式，程序执行过程中的意外突发行为也会导致 bug 的产生，具有五花八门的软件和交互界面的计算机网络经常是这种意外突发行为的来源。我们有时候讲“网络效应”，其中，某些未曾设想过的特征会突然出现，并有可能强化为有力的趋势。在某个应用程序以一种难以预料的方式投入使用时，这种情况很可能发生，垃圾邮件和钓鱼邮件就是大规模网络中邮件应用的一个意外突发行为。

这样的效应使得理解、预测和分析大规模网络中大型软件系统所出现的复杂行为变得非常困难。即使系统中的每一个部件都按照其设计参数运行，整个系统仍可能因为组件间不可预测的交互而产生不确定的结果。

复杂突发行为的出现也可能仅仅因为计算机是有限的。数字化信息在表示上常常包括细小的错误或误差，然而小错误经过数十亿步的计算后很有可能积聚成大灾难。一个非常具体的例子就是有限精度的浮点数运算，正如 2005 年数值计算会议上 William Kahan 在他的论文中展示的那样[1]，舍入误差以及对非常大或非常小的值的人工处理都可能导致灾难性的后果。

这些效应教会我们，让计算机代表我们来做事是一个不平凡的、高度智能化的尝试。因此，本书试图洞察一些基本计算原理，使得我们的方法和尝试能在最普遍意义上适应计算。

本书内容组织为 11 章，每一章的内容在与计算相关的活动中均起着重要作用。我认为这些内容在辅助“计算”上是调度和管理资源的焦点。计算即利用计算机和软件达到特定目标，这是一个故意模糊化的说法。在计算目标上，使一个电脑游戏工作和使一个分布式网络化的复杂金融交换系统工作是差不多的。而且，即使计算目标不同，在管理调度资源——信息表示、通信、计算单元、程序、内存、建模和分析等方面，一些具体的计算原理仍然可以帮助到设计师。本书的总体意图就是提出促使这些计算目标实现的基本原理，这一努力在广度和深度上均十分突出。

让计算如此有趣的一个因素是二进制表示的普适性。我们可以让比特位按照自己的想法代表任何事物，亦能以多种方式操作这些比特并从多种角度解释操作结果。就像把代数应用题转换为方程，根据直接的数学规则来获取初始方程的解一样，我们编写程序，并按照可以产生既定解释结果位的规则来操作比特位。大规模模拟、大数据以及复杂视觉渲染都具有这种属性，这些工作均可用来辅助我们理解并解释所操作的比特位。

我一直是“在高中、初中（甚至可能更早）阶段教授编程”的坚定支持者，原因之一在于编程对问题求解思维的系统训练。在编程这一活动中，学生需要分析问题，将问题分解为可管理的模块，想出程序需要怎样做才能解决问题（产生想要的结果），然后完成编程的整个过程：编写程序，利用已有的库，如果可能的话编译并运行程序，验证程序是否产生了想要的结果。最后一种训练（我们可称之为调试与验证的结合）是一种不仅仅适用于编程的技能。尽管我并不提倡每一个人都成为程序员，但学会良好编程中所用的技能是非常有价值的，因为这些技能对其他许多问题也广泛适用。

编程技能可以在处理复杂系统设计和分析时起作用，在这里我们接触到了 Denning 和 Martell 在“设计”这一章中强调的非常重要的一个领域。好的设计有许多有用的性质，这让我想起了一句名言——“整洁自有回报”，因为你可以在需要时找到整理好的东西。好的设计也自有回报，因为它有助于理解复杂性，并且具有通过演化、修正设计以达到新目标的能力。在互联网设计中，我们从先驱 ARPANET 那里汲取了教训，因为它无法在规模上扩展。然后我们设想了系统功能的分层并规范化层级间的接口，结果是在保持这些接口稳定的情况下，容许不同层在接口间实现与再实现的巨大灵活性。IP 协议是一个很好的例子，应用程序的设计者对于 IP 数据包如何运输（协议并未明确）一无所知，协议本身也不依赖于数据包载荷运输的是什么——载荷中比特位的含义是不清楚的。这种设计的一个结果是自 20 世纪 70 年代开始 IP 协议位于每一种新设计的通信系统的层级之上；另一个结果是新的应用程序不用改变网络就可放在互联网上，因为 IP 协议可将数据包传输给互联网上的软件。只有发送和接收的主机需要知道数据包运输的载荷比特位是什么意思，传输数据包的路由器也无需依赖数据包载荷的内容。

设计之于计算的作用怎样强调都不过分。不管是硬件、操作系统、应用程序、数据、文件目录结构或者语言的选择，归结到底都是思考设计以及设计的组合——系统怎样工作，虽然有时人们很少听说术语*系统工程*。我是一名系统设计师，乐于思考架构相关的问题：所有的部分如何组装到一起？每一部分应该是什么？设计如何适应新的需求？设计可维护吗？教会其他人如何设计困难吗？

对于好的设计，一个有趣的测试是看新接触某系统的人在不破坏之前设计功能的基础上能否让系统做一些新的事情。在某种程度上，这是对一个人理解程序或系统及其架构的一个十分有力的测试。你可能不需要了解系统的所有细节，但你需要了解足够多以确保你的改动不会产生预料之外的后果，这就是干净设计的含义，即设计能够合理地且安全地进行改动。我很高兴本书非常重视设计，并强调架构（不仅仅强调算法）对于设计的作用。

关于计算的基本原理还有许多可以说的，但那正是本书接下来的要点。记住这些原理将使计算系统的设计更加可控，继续阅读吧！

Vint Cerf

Woodhurst，VA

2014 年 4 月

前　言

Great Principles of Computing

就在70年前，除了少数专家之外，没有人听说过计算机。现在，计算机、软件和网络无处不在。在地球上的任何地方，它们都以更快的发展速度给我们的生活带来了各种各样的好处。

在这么短的几十年中，我们学会了设计和建造如此规模的系统，这真是一件令人吃惊的事。如今，通过支持大规模合作，计算技术使得知识工作能够自动化，同时也在不断扩大生产力。第二次机器革命正扑面而来[1]。这是如何实现的？是什么样的伟大思想使这一切成为可能？

计算机给我们带来好处的同时也带来忧虑。计算机带来的自动化是否会使很多工人失业？计算机是否会成为终极监督工具而使我们失去隐私？计算机是否会发展出超越人类的智能？计算机能做的事情会有限制吗？

我们相信，理解计算的原理和法则可以帮助人们理解计算是如何完成如此之多的工作的，并消除他们的忧虑。为此我们写了这本书，书中介绍了关于计算的一些最重要的原理，并以任何对计算有一定了解的人都能理解的方式呈现。

计算机科学（computer science）不只是设计计算设备的工程领域，它是一门关于信息处理的科学。计算受科学原理和法则支配，这些原理和法则告诉我们计算机能做什么、不能做什么。信息的法则揭示了根据物理法则无法直接得出的新的可能性和限制。专家们赋予计算机许多计算科学（computing science）告诉我们不可能拥有的能力，同时，这些专家又低估了计算机真正的能力。

计算机科学与很多其他领域相互交叉。许多科学与工程领域都有计算（computational）分支，如计算物理、计算化学、生物信息学、数字化产品设计与制造、计算社交网络、计算心脏病学等[2]。各层次的教育者正努力在他们拥挤的课程表中加入计算相关的课程，以保证课程体系的先进性。但仍有很多中学由于缺少计算机科学方面的教师而不能开设计算机课程。在商业领域，诸如“大数据”“云计算”“网络安全”等热门词汇也散发出共同的信号，期望“计算原理”在数据管理、分布式计算、信息保护中发挥作用。

一直以来，人们把计算看作一个按照摩尔定律高速发展的技术领域[3]。而我们的观点有所不同，我们相信计算更应该被描述为一个科学领域，具有跨越所有计算技术以及人工

或自然的信息处理的基本原理。我们需要一种新的方法来刻画计算。就像望远镜之于天文学、显微镜之于生物学，计算机是计算的工具，而非计算的研究对象。

本书的重要原理框架（great principles framework）就是这样一种新的方法。它将计算原理分为 6 个类别：通信、计算、协作、记忆（存储）、评估和设计。计算原理是指用来指导或约束我们如何操纵物质和能量来进行计算的声明。计算原理可以是：（1）重现，包括描述可重复的因果关系的定律、过程及方法；（2）行为准则。局部性原理（locality principle）就是重现的一个例子：每一个计算在一定的时间间隔内，其对数据的访问都聚集在一个小的子集里。行为准则的一个例子就是网络程序员将协议软件划分为多个层次。所有这些原理的目的，都是希望通过增进理解和降低复杂性从而得到良好的设计。

每种计算技术都利用了这些类别的原理。这个框架是广泛和全面的，覆盖了计算的每个部分，包括算法、系统和设计。

从事计算工作的人员形成了许多计算领域（computing domain）——实践社区，如人工智能、网络安全、云计算、大数据、图形学以及科学计算（computational science）等。这些领域都专注于推进领域向前发展并与其他社区互动，它们既从计算原理中获益，又受其约束。没有这些计算领域的原理框架是不完整的。

由于这 6 个类别过于庞大，我们决定将其所覆盖的范围分成 11 个更容易管理的模块，就像你在目录中看到的那样。关于这一点，我们将在第 1 章中详细说明。

从机器到通用的数字化

计算的机器是早期计算领域的关注中心（从 20 世纪 40 年代到 20 世纪 60 年代）。计算被看作机器执行复杂演算、解方程、破译密码、分析数据及管理业务流程的行为。那时的先驱们将计算机科学定义为研究以计算机为中心的各种现象。

然而，这些年来，这一定义变得越来越没有意义。20 世纪 80 年代的科学计算运动认为，计算是除了传统的理论和实验之外的一种新的做科学研究的方法。他们使用“计算思维”（computational thinking）这个术语作为研究和问题求解的思维训练，而不是作为建造计算机的方法。十年之后，一些领域的科学家开始发现各自领域内的自然信息处理，其中包括生物学（DNA 翻译）、物理学（量子信息[4]）、认知科学（脑力过程）、视觉（图像识别）和经济学（信息流）。计算的重点从机器转变到信息处理，包括人工信息和自然信息。

现在，随着几乎所有事物的数字化，计算进入了人们的日常生活，包括求解问题的新方法，艺术、音乐、电影的新形式，社交网络，云计算，电子商务，以及新的学习方法

等。用计算作比喻成为日常语言中的必要组成部分，比如“我的软件有反应了”或“我的大脑崩溃了需要重启”这样的表达方式。

为了应对这些变化，各大学一直都在设计新的基于原理的方法来开展关于“计算”的教学。华盛顿大学是这方面的先驱之一，它开发了关于熟练掌握信息技术的一门课程和教材，目前已经在高中和大学中广泛使用，以帮助学生学习并应用基本计算原理[5]。教育考试基金会（Educational Testing Foundation）与美国国家自然科学基金会（National Science Foundation）合作，开发了一门新的基于计算原理的先修课程[6]。现在很多人使用“计算思维”这个词，指的是在很多领域和日常生活中使用计算原理，而不仅局限于科学计算[7]。

随着计算领域的日趋成熟，它吸引了其他领域的众多追随者。我们知道有 16 本书是为感兴趣的非专业人员来解释计算的各个方面[8]。大部分书关注的只是单个部分的内容，如信息、编程、算法、自动化、隐私以及互联网原理等。本书则将这个领域作为一个整体来看待，给出所有各部分如何组合在一起的系统叙述。读者会发现在所有这些部分的背后是一套连贯的原理。

根据教授从其他专业转到计算机科学的研究生的经验，我们发现对于初学者来说，原理框架比技术框架更容易理解。当早期核心技术很少的时候，用技术思想的观点来描述该领域是一种好方法。1989 年，美国计算机协会（Association for Computing Machinery，ACM）列出了 9 大核心技术。而在 2005 年，ACM 列出了大约 14 种，到了 2013 年，则有约 18 种。本书的 6 类原理框架并不是重新定义计算的核心知识，但它确实提供了一种看待该领域并降低其表面复杂性的新方式。

起源和目标

我们经常被问及 6 类原理的起源。20 世纪 90 年代，本书作者之一 Peter J. Denning 在乔治梅森大学（George Mason University）开始这个项目。他从众多的同事那里收集了一个可能的原理陈述的列表。他发现了 7 个自然的群集，并将它们称为通信、计算、记忆（存储）、协作、评估、设计和自动化[9]。当组织本书的时候，我们意识到自动化并不是一个操纵物质和能量的类别，而是人工智能计算领域的重点。在本书中，我们从类别集合中删除了自动化，并将其包含在计算领域中。

这 6 个类别并不是把计算的知识空间划分成分离的片段。它们就像六角亭的窗户，每一扇窗户都以独特的方式呈现出内部空间，但同一件事物可以从多个窗户看到。例如，互联网有时以数据通信的方式、有时以协作的方式、有时以记忆（存储）的方式被看到。

这组类别有一个类别数目可控的框架，从而满足了我们的目标。虽然计算技术的列表还将继续增长，计算领域的集合也会扩大，但是类别的数目在较长时间内应该会保持稳定。

这本书是关于计算机科学的一个整体视角，注重最深入、最广泛的原理，即“宇宙普适的”原理（cosmic principle）[10]。本书将计算视作一种深层次的科学领域，其原理将影响包括商业和工业在内的其他每一个领域。

这本书是为所有想利用计算科学来达到其目标的人而设计的。受过科学教育的读者可以学到从算法到系统横跨整个领域的计算原理。而计算领域内的人，例如一个想要学习并行计算的程序员，可以找到这个巨大领域内不太熟悉的部分的概述。对于大学里学习诸如“计算机科学基础”课程的学生，本书可以帮助他们理解计算技术是如何影响他们的，例如网络和互联网如何使社交网络成为可能。初出茅庐的科学家、工程师和企业家可能在本书中找到一个面向整个计算机科学的科普型方法。

致谢

Peter 要感谢他在计算原理的漫长旅程中遇到的许多人，这一旅程开始于他 11 岁时，那时他的父亲给了他一本关于机器原理的不同寻常的书——1911 年出版的《 How It Works 》[11]。1960 年，他的高中数学老师和科学社团导师 Ralph Money，鼓励并引导他把精力投入计算机——引领未来的机器中去。1964 年，当他成为 MIT Mac 项目的学生时，他的导师 Jack Dennis、Robert Fano、Jerry Saltzer、Fernando Corbato 和 Allan Scherr 把他的兴趣扩展到所有计算背后的基本原理。1967 年他发表的第二篇论文是关于存储管理的工作集原理，其灵感主要来自于 Les Belady、Walter Kosinski、Brian Randell、Peter Neumann 和 Dick Karp 的帮助。1969 年，他带领一个工作小组设计操作系统原理的核心课程，他的队友 Jack Dennis、Butler Lampson、Nico Habermann、Dick Muntz 和 Dennis Tsichritzis 帮助确定了操作系统的原理，其中也包括 Bruce Arden、Bernie Galler、Saul Rosen 和 Sam Conte 的见解。在之后的几年里，Roger Needham 和 Maurice Wilkes 提供了关于操作系统原理的很多新的见解。1973 年，他与 Ed Coffman 合写了一本关于操作系统理论的书。

1975 年，Jeff Buzen 把他吸引到操作分析的新领域，这项研究关注计算机系统性能评估的基本原理。在那段时间里，Erol Gelenbe、Ken Sevcik、Dick Muntz、Leonard Kleinrock、Yon Bard、Martin Reiser 和 Mani Chandy 都促进了他对计算原理的理解。

1985 年，ACM 教育委员会（ACM Education Board）请他领导一个项目，以确定“计算”作为一门学科应具备的核心原理，用于设计 1991 年 ACM/IEEE 的课程推荐。他非常

感谢这个团队加深了他对计算原理的理解，这个团队的成员有：Douglas Comer、David Gries、Michael Mulder、Allen Tucker、Joe Turner 和 Paul Young。

20 世纪 90 年代中期，他开始在一个统一的框架下收集所有的计算原理。Jim Gray 嘱咐他去寻找“宇宙普适的”原理——那些有足够深度和广度的、在任何时间、在宇宙中任何部分都有效的原理。他设计了一门叫作“信息技术核心”的“旗舰”课程，并在乔治梅森大学的同事 Daniel Menascé、Mark Pullen、Bob Hazen 和 Jim Trefil 的帮助下于 1998 年启动了该课程。

2002 年，ACM 教育委员会邀请他成立一个重要原理工作组，以便就重要原理框架如何促进未来课程的设计给出建议。一个相当优秀的团队为此走到了一起，他们是：Robert Aiken、Gordon Bell、Fred Brooks、Fran Berman、Jeff Buzen、Boots Cassel、Vint Cerf、Fernando Corbato、Ed Feigenbaum、John Gorgone、Jim Gray、David Gries、David Harel、Juris Hartmanis、Lilian Israel、Anita Jones、Mitch Kapor、Alan Kay、Leonard Kleinrock、Richard LeBlanc、Peter Neumann、Paul Rosenbloom、Russ Schackelford、Mike Stonebraker、Andy Tanenbaum、Allen Tucker 和 Moshe Vardi。

关于计算中的科学问题，Rick Snodgrass（一个追求“ Ergalics”[⊖]的志同道合的人）给了我们很多睿智的建议。Vint Cerf 和 Rob Beverly 就网络相关的章节也给出了很多有用的建议。

Peter 最感谢妻子 Dorothy Denning，感谢她对于清晰的逻辑流程和坚实基础的一贯坚持，感谢她始终如一地鼓励他抓紧时间写作。同时也要感谢女儿 Anne Denning Schultz 和 Diana Denning LaVolpe 对他的信任和支持。

Craig Martell 之所以被吸引来合作写这本书，是因为他总是想要参数化这个世界。在这方面，计算作为一个领域呈现出特别迷人的挑战，因为它是科学、数学和工程学的融合体。他时常痴迷于这些机器居然能工作！

Craig 要感谢 Mitch Marcus，他们在宾夕法尼亚大学共同讲授了“信息技术及其对社会的影响”这门课程。制作这门课程教学大纲的过程开启了他对本书的贡献。他还要感谢合作者 Peter Denning，是他让整个写作过程变得有趣且富有成效。最后，他要感谢 Pranav Anand、Mark Gondree、Joel Young、Rob Beverly 和 Mathias Kölsch，他们使工作毫不沉闷。

Craig 对于本书的贡献离不开妻子 Chaliya 的耐心与全方位的支持，还有女儿 Katie 崇敬的笑容。有了她们，他坚信自己是这世界上最幸运的男人。

⊖ Ergalics 是关于计算工具以及计算本身的科学，是一项由美国亚利桑那大学 Rick Snodgrass 提出并开展研究的项目，详情请见 http://www.cs.arizona.edu/people/rts/ergalics/。——译者注

目 录

Great Principles of Computing

第1章
Great Principles of Computing

作为科学的计算

计算机科学研究计算机周边的各种现象。

——Newell，Simon 和 Perlis

计算机之于计算机科学，正如望远镜之于天文学。

——Edsger W. Dijkstra

计算与科学密不可分：计算不仅仅是一种数据分析的工具，更是一种用于思考和发现的方法。

这种观点的形成并得到广泛认同经历了曲折的过程。计算是一门相对年轻的学科，其作为一个学术研究领域确立于20世纪30年代，确立的主要标志是由Kurt Gödel（1934）、Alonzo Church（1936）、Emil Post（1936）、Alan Turing（1936）等人所发表的一组重要论文。这些研究者敏锐地意识到了自动计算的重要性，为“计算”这个概念奠定了必要的数学基础，并探讨了实现自动计算的不同模型。但人们很快就发现，这些看似不同的计算模型实际上是等价的，即由一种模型所描述的计算过程总能够被其他模型所描述。而更为值得关注的是，这些不同的计算模型都指向了一个共同的结论：某些具有现实意义的问题（例如一个算法是否能够终止）并不能通过计算的方式进行求解。

在这些研究者所处的时代，“计算”和“计算机”这两个术语已经得到了广泛的使用。那时，计算指的是求解数学函数的机械步骤，计算机则指的是执行计算的人。作为对这些研究者所开创的崭新领域的致敬，第一代数字计算机的设计者通常会将其所构造的系统命名为一个带有后缀字符串“AC”的名字，其含义为自动计算机（automatic computer）或类似的变体。这些名字的典型代表包括ENIAC、UNIVAC、EDVAC、EDSAC等。

在第二次世界大战的初期，美、英两国的军方开始对自动计算机产生兴趣，他们想使用自动计算机进行弹道计算和密码破译。为此，他们资助了若干项目进行电子计算机的设计与实现。在二战结束前，只有一个项目顺利完成。这是一个设在英国布莱切利公园的绝密项目。这个项目使用由阿兰·图灵设计的方法成功破译了德军使用的Enigma密码系统。

在20世纪50年代初，很多参与这些项目的研究人员开始创办计算机公司。20世纪

50年代末，大学也开始进行计算机领域的研究项目。从此以后，计算机领域的学术研究和工业实践开始持续地增长，直到我们目前所看到的一个现代化的庞然大物。这个庞然大物包含了海量的互联网连接和数不清的数据中心，据说消耗了全世界所生产的3%的电能。

图1.1 查理斯·巴贝奇（Charles Babbage，1791—1871，左图）发明了差分机，一种机械式的算术运算机器。后来，他提出了分析机的设计方案：该方案如果能够实现，则有望成为一种通用计算机。Ada Lovelace（1815—1852，右图）帮助Charles Babbage完成了分析机的设计。Lovelace的视野不仅仅局限在数字计算上，而是扩展到了谱曲、绘画，甚至逻辑推理等领域。在1843年，她编写了一段能够使用分析机计算伯努利数列的程序。她也因此被称为世界上“第一个计算机程序员”（来源：Wikipedia Creative Commons）

2

在其青年时期，计算对于已有的科学和工程领域而言，是一种难以被理解的新兴事物。对于不同的观察者而言，第一眼看上去，计算像是一种对数学、电子工程或其他科学领域的技术应用（译者注：即，计算来源于这些领域，没有这些领域，计算就如同无源之水）。经过多年的发展以后，计算似乎又为人们提供了无数的深刻见解，打破了人们对计算的早期理解，即计算不可能成为一个独立的领域，而必将被其来源领域所重新吸纳。到1980年，计算领域对于算法、数据结构、数值方法、编程语言、操作系统、网络、数据库、图形图像、人工智能、软件工程等方面已经具备了相当成熟的理解。计算领域的伟大技术成就——芯片、个人计算机、互联网——将计算带入了更加丰富多彩的境界，并促成

了更多新兴子领域的产生，包括网络科学、web 科学、移动计算、企业计算、协同工作、网络安全、用户界面设计、信息可视化等。由此产生的大规模商业化应用为社交网络、演化计算、音乐、视频、数字化摄影、视觉、大规模多用户在线游戏、用户生成内容（user-generated content）以及其他领域带来了全新的研究挑战。

为了适应这种持续的变化，计算领域的名称在历史上已经发生了数次变化。在 20 世纪 40 年代，这个领域被称为自动计算。在 20 世纪 50 年代则被称为信息处理。在 20 世纪 60 年代，其进入了学术界，那时，美国人将其称为计算机科学，欧洲人则称其为信息学。到 20 世纪 80 年代，计算领域已经包含了一组具有复杂相关性的子领域，包括计算机科学、信息学、计算科学、计算机工程、软件工程、信息系统、信息技术等。到 1990 年，“计算”这个词已经成为引用这些领域的标准术语。

图 1.2 阿兰·图灵（Alan Turing，1912—1954）认为计算就是对数学函数的求解。1936 年，他设计出一种后来被称为图灵机（Turing Machine）的抽象计算机器。这个机器由一条无限长的纸带和一个可沿该纸带左右移动的有限状态控制器构成。纸带上包含了无限个左右排列的方格，每个方格里可以存放来自特定符号集合的一个符号。控制器每次可以向左或向右移动一个方格。在每一次移动前，控制器读取当前方格内存储的符号，并有可能在当前方格中写入新的符号，然后向左或向右移动一个方格，从而进入一个新的控制状态。图灵展示了如何构造一个统一的计算机器对任何计算过程进行自动执行。图灵认为，任何具有可计算性的函数都可以被一个具体的图灵机自动计算。图灵也发现存在一些不可计算的函数，例如：确定一个具体的图灵机是否会停机或陷入无限循环（来源：Wikipedia Creative Commons）

计算机科学作为一个正式的学术领域确立于1962年：这一年，美国的普渡和斯坦福两所大学成立了计算机科学系。在当时，这个领域的领导者不仅需要时刻向人们解释这个领域到底是做什么的，还需要不断抵御来自保守评论家的苛责。这些评论家认为，排除电子工程和数学的内容，计算机科学并无实质性的新东西。在1967年，Allen Newell、Alan Perlis、Herbert Simon对这种观点给出了如下回应：计算机科学是一门全方面研究“计算机周边现象”的科学。但是，许多传统的科学家则反对“这个领域是一门科学”的观点。他们认为，真正的科学关注于那些在自然（自然过程）中发生的现象，而计算机仅仅是一种人造物，不属于自然的范畴。诺贝尔经济学奖得主Simon强烈反对这种对于科学的“自然解释”。为此，两年后，Simon出版了一本名为《The Sciences of the Artificial》的书。在这本书中，Simon指出：除了“以自然过程为研究对象”这条准则之外，经济学和计算机科学符合科学的其他所有既有准则，因此，称其为科学并无不妥。当然，为了区别于传统的自然科学（natural science），可以将这类科学称为人工或人造科学（artificial science）。

3
~
4

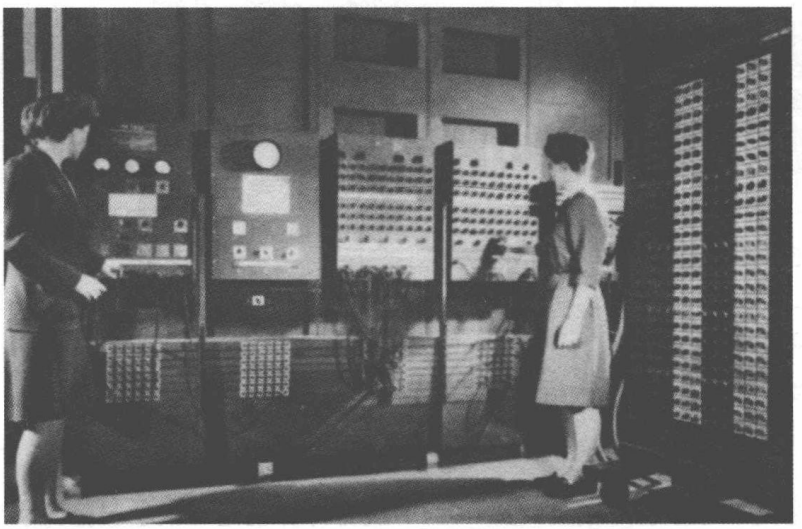


图1.3 在Babbage尝试构造一个可以实际运行的分析机的努力失败之后的80年内，没人再试图去设计通用的计算机器。在20世纪30年代末，美、英两国的军方开始尝试采用电子机器进行弹道计算和密码破译。1944年，美国军方资助的ENIAC计算机开始正式运转。这台计算机坐落在宾夕法尼亚大学。这个项目由John Mauchly和J. Presper Eckert领导。ENIAC上的第一批程序员包括Kay McNulty、Betty Jennings、Betty Snyder、Marlyn Wescoff、Fran Bilas、Ruth Lichterman等人。这张图展示了Jennings（左）和Bilas（右）正在操控ENIAC的主控制面板。在那个时候，计算机指的是人，而计算就是这些人的专职工作；电子机器则被称为自动或电子计算机。程序设计的主要活动是在不同的插座之间进行接线（来源：宾夕法尼亚大学）

计算的范型

1962 年之后的 30 年中，传统自然科学的研究者经常对“计算机科学”这个名字进行质疑。在计算机科学这个新兴领域中，他们可以很轻易地看到工程范型（系统设计与实现）和数学范型（定理证明），但是却很难看到太多的科学范型。此外，与 Simon 的辩解相反，这些研究者认为科学是一种关于自然世界的方法，而计算机则更像是一种人工制品。 5

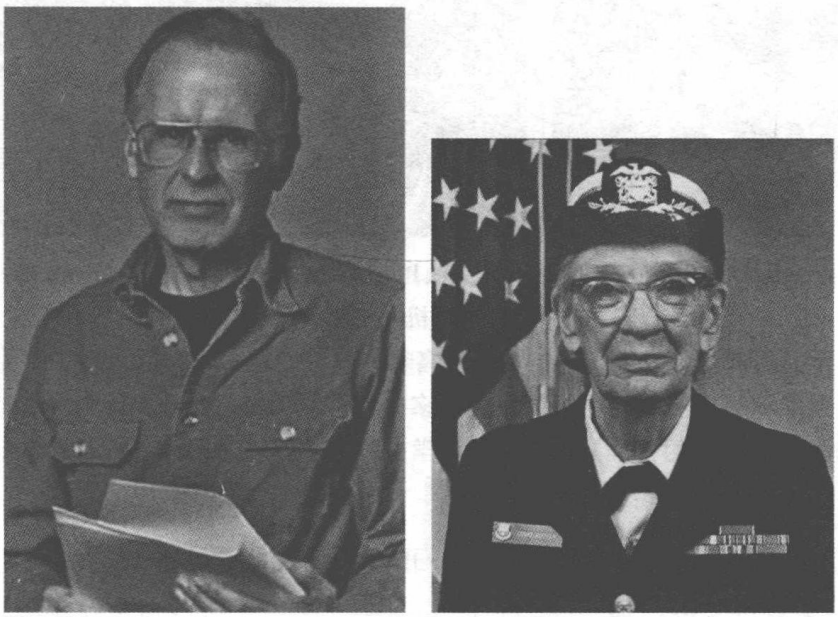


图 1.4 两位先驱者 John Backus（1924—2007，左图）和 Grace Hopper（1902—1992，右图）设计出了高级程序设计语言：通过编译器，高级程序设计语言可以被自动转换为机器代码。1957 年，Backus 领导的团队开发出了 FORTRAN 语言：一种面向数值计算的高级程序设计语言。1959 年，Hopper 领导的团队开发出了 COBOL 语言：一种面向商业计算的高级程序设计语言。这两种语言至今仍然被使用。这些发明使得类似 ENIAC 的接线式程序设计彻底失去了生存空间，并且极大地降低了计算领域的入门门槛。自此以后，人们已经发明了数千种的程序设计语言（来源：Wikipedia Creative Commons）

这个领域的创立来自这三种不同的范型（工程范型、数学范型和科学范型）[1]。一些人认为计算是应用数学的一个分支，另一些人则认为计算是电子工程的一个分支，其他人则认为计算是面向计算的科学的一个分支。在第一个 40 年中，这个领域主要关注于工程：如何构造可靠的计算机及其网络以及如何开发复杂的计算机软件，吸引了几乎每一个人的注意力。到 20 世纪 80 年代，这些工程问题基本上已经得到了有效的解决；同时，在计算机网络、超级计算机、个人计算机的帮助下，计算开始向各个领域快速渗透。在 20 世纪 80 年代，计算机的能力已经变得异常强大，而那些具有远见卓识的科学家开始意识到 6

计算机可以用来解决科学和工程中的重大挑战问题，并由此产生了“计算科学”的研究运动。许多国家的科学家共同参与到这一运动中，并最终使得美国国会在1991年通过了名为“高性能计算和通信”（High-Performance Computing and Communications，HPCC）的行动计划，该计划旨在探索采用计算方法解决科学和工程中的重大挑战。

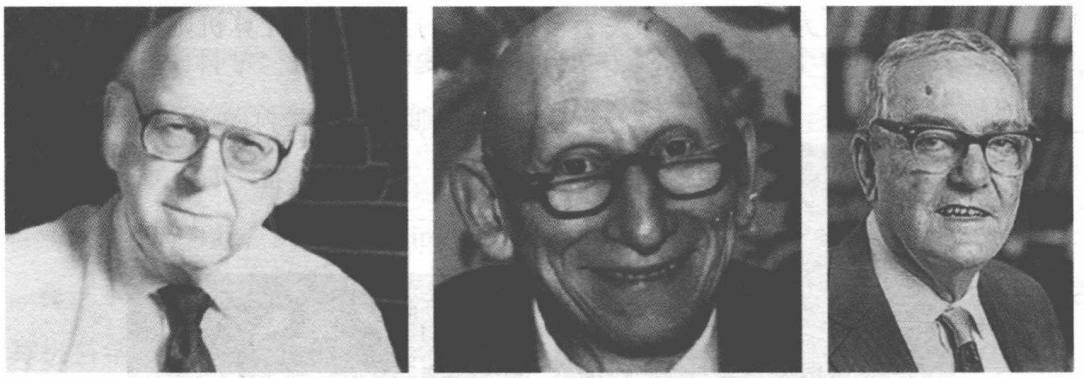


图1.5 Allen Newell（1927—1992，左图）、Alan Perlis（1922—1990，中图）、Herb Simon（1916—2001，右图）认为计算是一种关于计算机周边各种现象的科学。1967年，他们认为，计算机科学是研究计算相关事物的一种必备科学，涉及计算机器、软件、智能、信息、系统设计、图形、解决实际问题的算法等众多方面。Simon则进一步认为，人工科学（即研究人造制品周边现象的科学）如同传统科学一样，也是一种科学（来源：Wikipedia Creative Commons）

目前，人们看起来已经形成了这样一种共识：计算是一种典型的科学和工程；而且，无论是科学还是工程自身，都不能有效刻画计算。那么，如何刻画计算，计算的范型是什么？

从一开始，这个领域的领导者就一直被范型问题所困扰。从发展历程来看，目前已经出现了试图统一各种不同视角的三次浪潮。第一次浪潮，由Newell、Perlis、Simon三人（1967）领导，认为计算不同于所有其他科学领域，原因在于计算的研究对象是信息处理过程。Simon将计算称为人工科学（1969），其背后所基于的一个共识性观念是，计算不是一种自然过程。这一波浪潮的流行语是：“计算研究计算机周边的各种现象。”

第二次浪潮关注于程序设计，即如何进行算法设计，从而产生有用的信息处理过程。
7 在20世纪70年代早期，两位计算领域的先驱者Edsger Dijkstra和Donald Knuth就强烈主张将算法分析作为计算的统一范型。这一波浪潮的流行语是：“计算机科学就是程序设计。”这种观点近来已经逐渐落没，因为这个领域的发展已经远远超过程序设计的范畴，同时也因为公众对于“程序员”这一概念的理解变得过于狭隘（仅仅是一个编写程序源代码的人）。

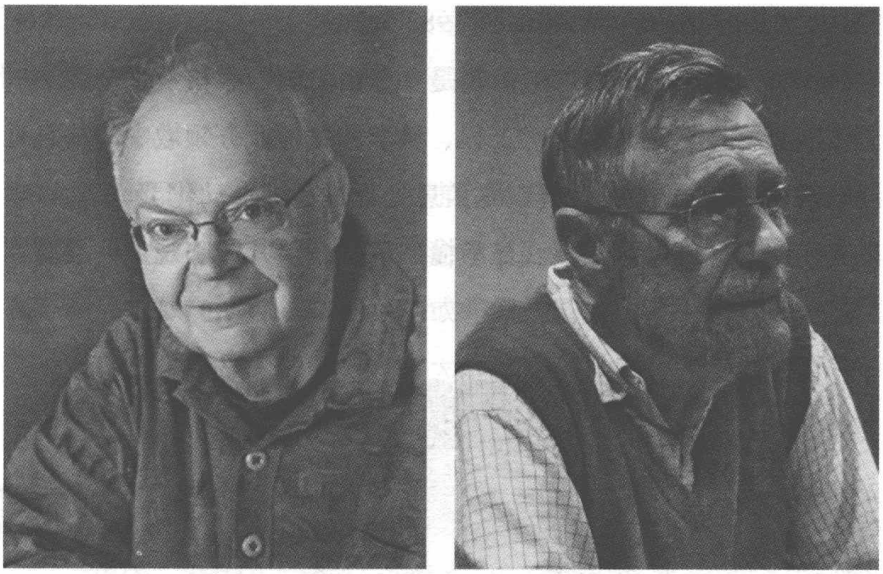


图 1.6 Donald Knuth（1938—今，左图）和 Edsger Dijkstra（1930—2002，右图）认为程序设计是计算的核心。1970 年左右，他们认为算法的设计与分析过程应该是计算机科学的核心关注点。对他们而言，一个程序设计专家就是一个计算机科学家。不幸的是，这种重要的观点在 20 世纪 90 年代后期不复存在，因为程序设计人员被官方定义为低层次的源代码编写者（来源：Wikipedia Creative Commons）

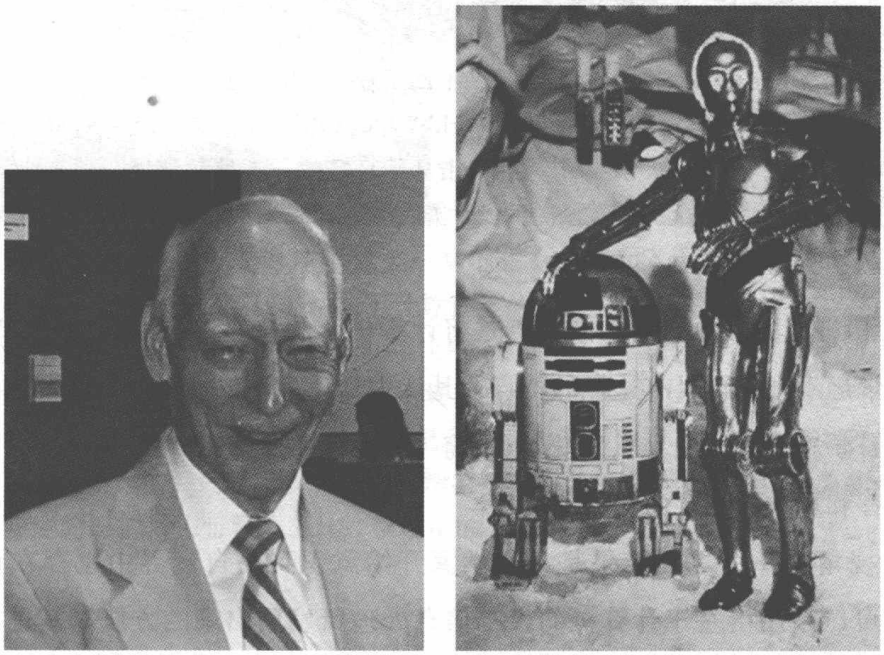


图 1.7 计算机系统领域的先驱 Bruce Arden（1927—今，左图），在 COSERS 项目中形成了“计算不仅仅是程序设计”的观点。在 20 世纪 70 年代末，他领导的团队将计算领域定义为一个对“什么可以被自动化”进行研究的领域。在当时，这种观点非常迎合公众对于机器人（如右图中出现在电影《星球大战》中的两个机器人）的好感。但这种观点并没有生存太长时间，因为在短短几年之后人们对科学的理解发生了重大的变化（来源：Wikipedia Creative Commons）

第三次浪潮则源于由 Bruce Arden（1983）所领导的项目：Computer Science and Engineering Research Study（COSERS）。这是一个由美国国家科学基金在 20 世纪 70 年代资助的项目。该项目将计算定义为：对工程、科学和各种业务领域中的信息处理过程的自动化。这一波浪潮的流行语是："计算就是信息处理过程的自动化。"虽然该项目的结题报告对很多深奥难懂的研究问题进行了浅显易懂的解释，但其核心观点并没有在大众中得到广泛传播。

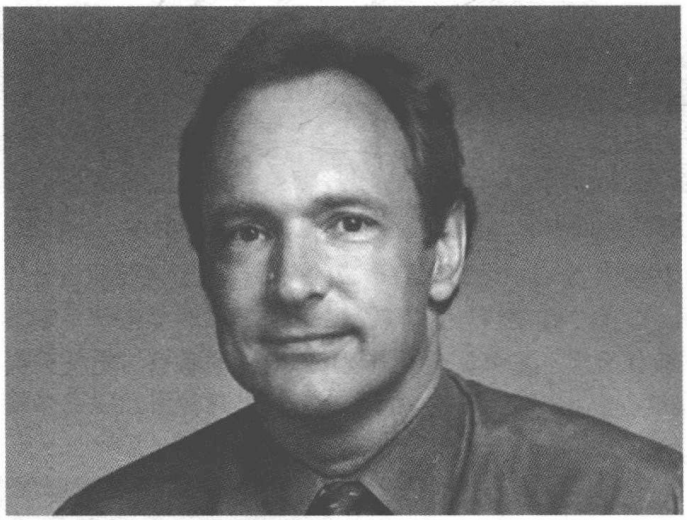


图 1.8 Tim Berners-Lee（1955—今）展示了一种不同于普遍将计算理解为"由一组机器构成的网络"的视角。1989 年，他发明了万维网（World Wide Web）：在万维网上，存储于机器中的信息之间相互链接，并且可以随着鼠标的点击从信息链的一端移动到另一端。他认为这种由信息链接形成的网络中蕴含了大量的新型计算行为，使得人们能够为信息赋予更多的含义（来源：Wikipedia Creative Commons）

这三次浪潮所具有的一个共同特点是将计算机作为核心关注点。始于 20 世纪 80 年代的计算科学运动则不再具有这样的特点：其认为，计算不仅仅是科学研究的工具，而更是一种进行科学思考和科学发现的崭新方法。计算科学的拥护者将计算视为帮助他们理解信息处理过程和算法控制能力的得力伙伴。

这种新视角的一个重要结果是：科学家开始意识到在自然界中也存在信息处理过程，并且也可以采用由基于计算机的人工信息处理所发展出来的方法对其进行研究。生物学是其中的一个典型代表：作为对认知科学家 Douglas Hofstadter（1985）观点的一种呼应，诺贝尔奖获得者 David Baltimore（2001）认为，生物学已经变为一种信息科学。David Bacon（2010）认为物理学也正在发生类似的变化：作为量子计算的支撑理论，量子力学也是一种信息科学。Erol Gelenbe（2011）列举出了一长串的科学领域，

其主要研究对象都涉及自然界的信息处理。计算机科学的方法同样适用于自然界的信息处理。这一结论进一步巩固和强化了 Herb Simon（1969）“计算机科学的确是科学”的观点。

最近，Paul Rosenbloom（2012）注意到了另外两点原因，使得“所有的计算都是人工过程”的观点变得愈发陈旧。第一，许多科学家开始认同，人也是全球生态系统的一部分，因此，人工制品也如同河狸筑造的水坝或蚂蚁构造的巢穴一样自然。第二，人类能够在任意粒度层次上修改自然过程的能力，极大地模糊了自然和人工的边界，例如干细胞克隆器官、有机生长的纳米机器以及转基因农作物等。

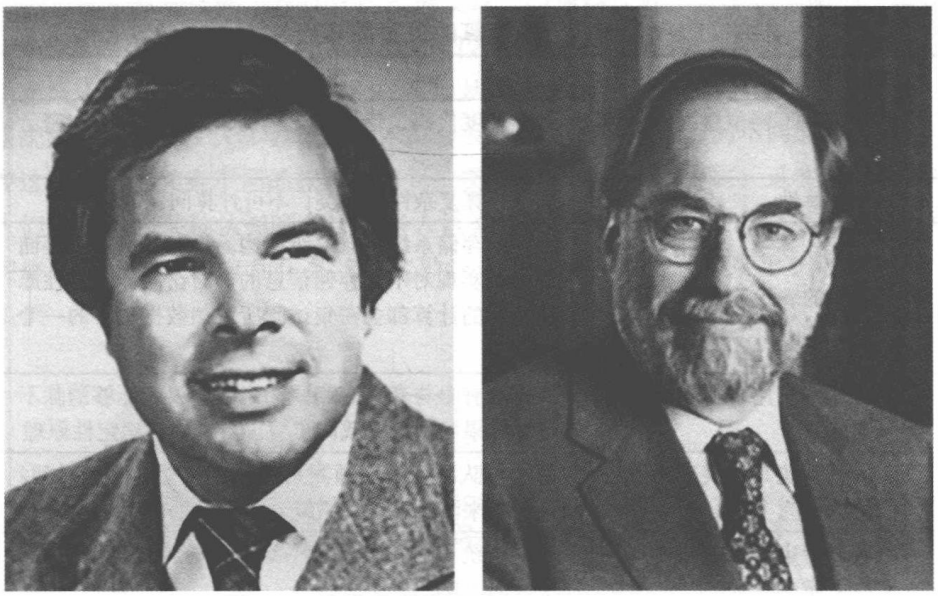


图 1.9 两位诺贝尔奖获得者，物理学家 Ken Wilson（1936—2013，左图）和生物学家 David Baltimore（1938—今，右图）站在了计算科学的前沿：他们认为计算是一种进行科学思考和科学发现的崭新方法。20 世纪 80 年代中期，Wilson 指出科学中的一些重大挑战问题可以通过计算得到解决，并认为应当使用具有高度并行性的超级计算机来进行这些计算。20 世纪 90 年代，Baltimore 指出，生物学已经变成对细胞和所有生命过程中所蕴涵的信息处理过程进行研究的一门学科。计算机科学家一开始并不愿意参与其中，但却对计算科学表达了坚定的拥护，从而导致了计算领域中的一场科学复兴运动（来源：Wikipedia Creative Commons）

计算的重要原理

对计算这一概念的理解的逐渐成熟，使得我们能不断从新的视角去确定计算领域的内涵。直到 20 世纪 90 年代，计算领域中绝大多数的科学家对该领域的理解都会落实到其所包含的一组核心技术上，例如算法、数据结构、数字化方法、编程语言、操

作系统、网络、数据库、图形图像、人工智能、软件工程等。这是对计算领域的一种深刻的技术型解释。本书对计算的理解则更加关注这些技术的能力和局限背后所隐藏的基本原理。

本书所给出的计算基本原理划分为6类：通信（communication）、计算（computation）、协作（coordination）、记忆（存储）（recollection）、评估（evaluation）、设计（design）（Denning 2003，Denning and Martell 2004）[2]（见图1.10）。这6类基本原理都关注如何通过操纵物质和能量来实现所期望的计算。表1.1对这6类基本原理分别进行了简要描述，并给出了与之对应的篇章。

10
~
11

表1.1 计算的重要原理

类 别	关注点	示 例	核心章节
通信	信息在不同位置之间的可靠传输	最小长度代码，错误修复代码，文件压缩，加密/解密	3，11
计算	可计算性	问题计算复杂性的分类，不可计算问题的特点	4，5，6
记忆	信息的表示、存放与读取	所有的存储系统具有层级结构。没有任何一个存储系统能够实现对不同存储信息的等时访问。局部性原理：所有的计算都会密集访问所依赖数据集中的一个子集	7，11
协作	有效地利用多个自主的计算实体	使得所有参与者具有相同知识的协议，能够消除不确定性结果的协议，或同步协议。选择不确定性原理	2，8，9
评估	度量系统是否表现出预期的计算行为	采用排队网络模型预测系统的吞吐量和响应时间。设计实验来测试算法和系统	9，10
设计	通过特定结构的软件系统实现可靠性	复杂系统可以被分解为一组交互的模块和虚拟机。模块之间可以形成层级结构	10

12

计算的内涵不仅仅是一种基本原理或是在此基础上形成的一些核心技术。计算的内涵还和各种实践领域紧密相关（见图1.11）。除了计算基本原理的知识之外，计算领域的专业人员还需要具备4种核心的实践能力：程序设计、系统思维、建模以及计算思维。实践能力是通过长期的经验积累而形成一种技巧。实践者的技巧可以被分为如下几类：初学者（beginner）、进阶初学者（advanced beginner）、初级资质者（competent）、熟练工（proficient）、专家（expert）。例如，一个入门级的程序员可能会主要关注语法和编译问题以及bug的查找问题；一个专家级的程序员则能够构建大型的软件系统，解决复杂的系统问题，或对下级程序员进行指导。基本原理和实践相互交织在一起。人们通过体现出高超技巧的行动将基本原理应用于实践中，而新的基本原理又会从大量的实践中逐渐浮现出来。

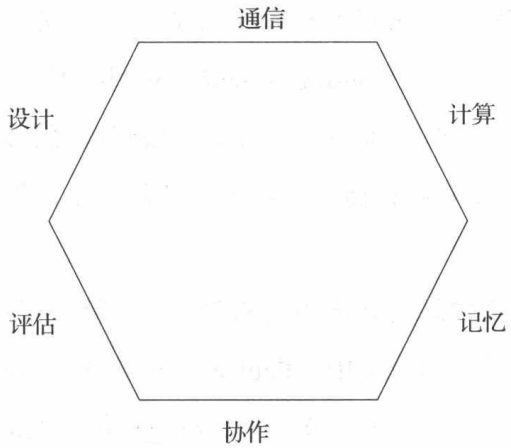


图 1.10 每一类基本原理反映了对计算的一种视角，即观察计算领域知识空间的一扇窗口。（对这 6 类基本原理的排列顺序并不反映它们之间的相对重要性。）同时，这 6 类基本原理也不是完全不相交的。例如，互联网既可以从通信系统的角度理解，也可以从协作系统或记忆系统的角度理解。大多数计算技术都涉及对这 6 类基本原理的不同组合：每一类基本原理在这种组合中具有不同的权重，但每一类基本原理都确实存在。这些基本原理类别表现了人们对计算的某种认知视角。一些人认为计算仅仅就是计算，其他人则认为计算是数据、网络化的协作或者自动化系统。这种划分框架能够在某种程度上扩展人们对计算本质的认识

13

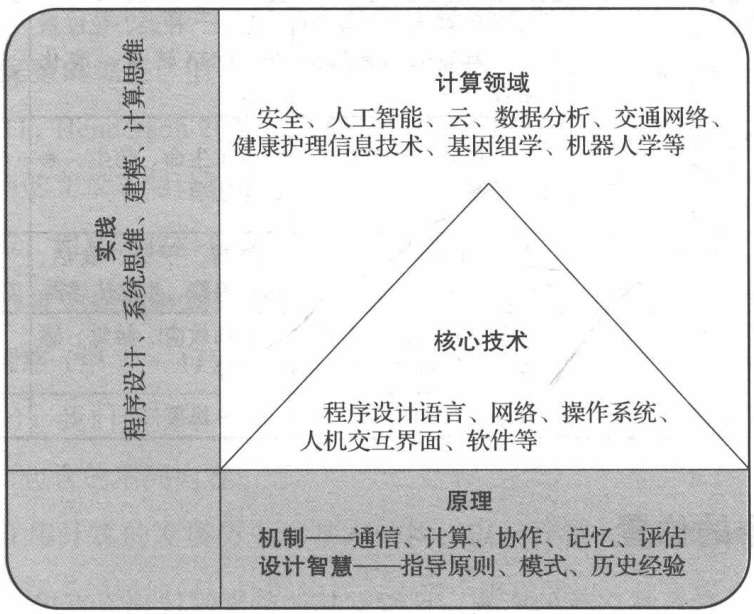


图 1.11 计算领域的发展建立在基本原理和实践两者的共同作用下。核心技术是被实践者在各种计算领域中广泛应用的工具。本书主要关注基本原理及其在若干关键领域的使用，而不涉及核心技术及其在实践中的应用。基本原理或者是一种机理——定律和重复出现的事物，或者是一种设计智慧——通过不断的积累形成的关于方法可行性的知识，从而使得构造可信、可靠、可用、安全的计算系统成为可能

计算的技术人员及其客户形成的社区称为计算领域。现实中存在数量众多的计算领域。ACM（the Association for Computing Machinery）给出了其成员所关注的至少42种专业领域（Denning 2001，2011），而更多的计算领域则被冠名以“计算应用”（computing applications）。下一章将简要介绍目前受到高度关注的4个计算领域：信息安全、人工智能、云计算以及大数据。

大多数计算领域与计算之外的其他领域相关。在一篇关于计算与物理学、生命科学、
14 社会学三个科学领域相互关系的论文中，Paul Rosenbloom发现了两种类型的关系：实现（implementation）和影响（influence）（Rosenbloom 2004，Denning和Rosenbloom 2009，Rosenbloom 2012）。实现指一个领域中的事物被用于构造另一个领域中的事物。影响指一个领域中的事物影响了另一个领域中事物的行为。这两种关系可以是单向或双向的。Rosenbloom通过一张表（表1.2）来说明计算与物理学、生命科学、社会学以及自身之间可能存在的丰富关系。其中计算与其自身的关系是因为通过不同计算领域之间的交互，计算会不断地实现或影响自身。

表1.2 计算与不同领域之间的交互示例

	物理学	社会学	生命科学	计　算
计算被实现	机械、光、电子、量子、化学过程	机械机器人、人类的认知、具有输入和输出的游戏	基因、神经、免疫系统、DNA转录、演化计算	编译器、操作系统、模拟器、抽象、过程
计算实现	建模、模拟、数据库、数据系统、量子密码学、3D打印	人工智能、认知建模、自主系统	人工生命、仿生、系统生物学	体系结构、语言
计算被影响	传感器、扫描仪、计算机视觉、文本识别	学习、程序设计、用户建模、授权、语音理解	眼睛、手势、表达、运动追踪、生物传感器	网络、信息保护与安全、并行计算
计算影响	运动、制造、操纵、开环控制	屏幕、打印机、图像、语音生成、网络科学	生物效应、触觉、感觉沉浸	分布式系统、网格
双向影响	机器人、闭环控制	人机交互、游戏	脑－机接口	

计算在科学中的位置

由于计算对科学存在广泛的影响，且所有其他科学领域都不直接与信息相关，所以Rosenbloom得出的结论是计算可以被认为是一种全新的科学领域。

计算领域对于信息处理的独特性是什么？信息在传统上指的是那些一旦被传播就会产生知识增加的事实。信息是一个古老的概念，哲学、数学、商业、人文学科、科学已经对信息进行了几个世纪的研究。科学关注于发现事实，通过事实形成模型，使用模型做出

预测，并将有效的预测模型转变为技术。科学家则将所有学习到的东西组织成“科学知识体系”。显然，信息在各种科学领域中都具有非常重要的角色。

与其他科学领域与信息的关系相比，计算领域与信息的关系体现出两个显著不同的特点。首先，计算领域强调对信息的变换，而不仅仅是对信息的发现、分类、存储或通信。算法不仅仅读取具有特定结构的信息，还会进一步修改信息的结构。进一步而言，人类也在不断地修改信息的结构（例如在互联网上），只不过我们还不知道人类对信息进行变换的计算模型是什么。纯粹的解析方法并不能帮助我们理解信息结构发生变化的动态机理。实验方法对于这一问题也缺乏有效的解决方案。

15
~
16

第二，计算不仅仅是描述性的，而更是生成式的。算法不仅仅描述了解决一个问题的方法，也会使得一台计算机去真正地解决这个问题。计算科学是唯一的对信息与活动之间的因果关系如此强调的一个领域。其他科学领域对信息则没有这样的视角。计算及其结果对各个科学领域产生了深远的影响。计算绝不仅仅是物理学、生命科学或社会学的一个子集。计算有资格成为一个独立的科学领域。

本书的关注点

计算已经得到了长足的发展，覆盖了非常丰富的研究内容，因此不可能在一本书中对计算进行一个完整的综述。有三本书可以被认为是“计算机科学百科全书”（Ralston 2003，Abrams 2011，Henderson 2008），这三本书的厚度分别是 2030 页、770 页和 580 页，这些书通过一系列的文章来对计算进行综述。在本书中，我们不试图对计算的研究内容给出一个全面的覆盖；相反，我们只会给出一组具有代表性的关于计算的重要原理。

这些代表性的重要原理被划分为 9 章：信息、机器、程序设计、计算、存储、并行、排队、设计以及网络（第 3 ~ 11 章）。前文所提及的计算的 6 个领域，每一个领域至少对应一章（见表 1.1）。我们期望本书的内容能够提供一种具有一定广度和深度的系统性视角，去理解计算所包含的不同内容。

第 1 章主要介绍计算的发展历史和基本结构，以及计算与其他领域的关系。第 2 章主要介绍计算的不同子领域如何从计算的基本原理中汲取知识，信息安全、人工智能、云计算以及大数据是典型代表。

第 3 章关注信息的本质，在信息之上计算机所能展现出的能力，以及计算机如何向其用户提供有意义的信息。第 4 章探讨计算机的构造技术，使得编写的程序能够控制电子线路去执行人类期望的计算。第 5 章关注程序设计，针对特定的问题设计相应的计算解决

方案的技巧，以及如何将程序转换为等价的机器代码。

第6章关注计算自身：一些问题能被快速算法求解，一些问题能被速度较慢的算法求解，而还有一些问题根本无法被计算机求解。第7章关注存储，即如何实现有效地信息存储与读取。

第8章探讨并行：通过一组相互协作的计算机并行工作，提高问题求解的速度。第9
17 章关注队列：在服务器集群为大规模并发请求提供服务时，一种预测系统吞吐量和响应时间的方法。

第10章关注设计：如何规划和组织可靠、可用、安全的计算系统。第11章以互联网为实例展示如何利用各种基本原理构造一个可靠的大规模数据通信网络。

本书的最后附上了参考文献目录，其中包含了一些给我们带来启发的代表性文献（不是对历史文献的完整性总结）。如果你在本书中发现了一个人名，那么你会在参考文献目录中至少发现一篇以此人为作者的文献。

总结

计算正在变得愈发成熟，展示出其在科学、工程和数学等方面的优良特性。计算的科学本质对于该领域的发展具有至关重要的作用：很多系统过于复杂以至于只能借助于试验方法对其进行研究。计算的覆盖范围相当广泛，包括各种自然或人工的信息处理过程。

本书揭示了所有计算过程所基于的一组基本原理。这些基本原理为众多计算领域以及物理学、生命科学以及社会学中的众多子领域提供了理论基础。

计算不是物理学、生命科学或社会学的一个子集。这些科学领域并不关注信息处理与变换的本质是什么。这种关注在所有其他科学领域中都不具有基础性地位。因此，计算有资格成为一个独立且重要的科学领域。

致谢

本章改编自“Great Principles of Computing”（Peter J. Denning，*American Scientist* 98（Sep–Oct 2010），369–372）。重新发表在《Best Writings on Mathematics 2010》（Mircea Pitici，Princeton University Press（2011））。本章内容的使用得到了《美国科学家》杂志
18 的授权。

第 2 章
Great Principles of Computing

计算领域

生物学是一种信息科学。

——David Baltimore

除了理论和实验之外，计算是进行科学研究的第三种方式。

——Kenneth Wilson

科学与科学应用密不可分，如同一个树上结出的多枚果实。

——Louis Pasteur

计算活动由人类实施，而不是基本原理。在长期的实践活动中，人们的计算活动逐渐形成了丰富多样的计算领域（computing domain）。每一个计算领域主要关注一项技术或其应用。例如，信息安全领域主要关注信息安全技术，而隐私领域则主要关注如何应用信息安全技术来保护个人的隐私信息。这些领域中的实践者分享相似的问题、技巧、方法，享受计算的基本原理带给他们的权利，同时也受到这些原理的限制。本书所阐述的计算的重要原理不可能脱离这些计算领域而独立存在（Rosenbloom 2012）（见图 2.1）。

*空气动力学数字仿真*是计算领域的一个实例。为了更有效地设计飞机，计算机科学家和航空领域的专家进行了长期的协同工作。自 20 世纪 80 年代以来，飞机制造公司开始使用数字仿真技术来设计机翼和机身。传统方法通过风洞和样机进行机翼和机身的设计，对于大尺寸复杂飞机的设计已经不具有可行性。通过运行在大规模并行超级计算机上的新型算法，工程师已经可以在不经过风洞试验的情况下设计出可以安全飞行的飞机。波音 777 是第一种完全通过数字化设计产生的飞机。航空专家和计算机专家紧密合作，产生了一个新的科学领域——计算流体力学，来计算气流的复杂运动。他们基于 3D 网格设计出相应的计算方法来求解机身周围空气的流体力学方程。他们探索出一种快速多重网格算法，能够基于超立方体并行处理器网络在很短的时间内完成大尺寸机身的设计工作（Chan and Saad 1986，Denning 1987）。他们还设计了动态网格精化方法，来提高气压和流速变化剧烈区域的计算精度。其中的有些方法甚至体现出了全新的计算基本原理。基于这些进展，计算方法已经成为流体力学不可缺少的构成成分。

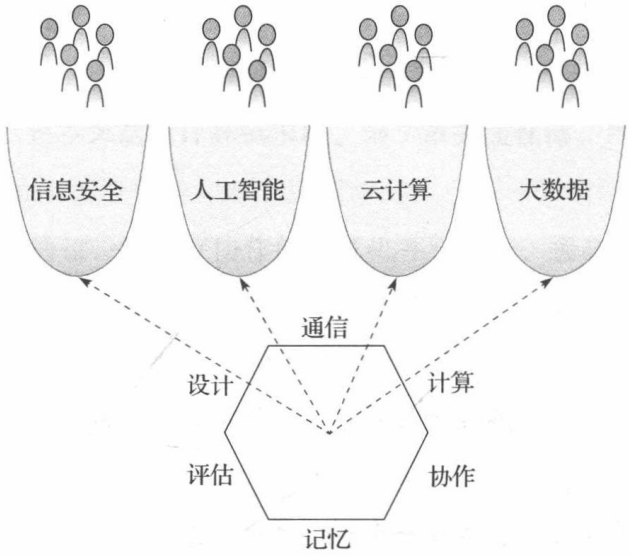


图 2.1 图中底部所示的 6 种类型的计算原理都关注于通过管理物质和能量来产生预期的计算行为。而图中上部的计算领域则是实践领域。这些实践领域中的人们通过灵活应用计算的基本原理来求解他们遇到的各种问题（带箭头的虚线）。这些实践领域中的工作为计算也为自身探索出新的基本原理

20

目前已经发展了丰富多样的计算领域。ACM（Association for Computing Machinery）总结出其成员所关注的 42 种计算领域，以及数十种的相关的领域（Denning and Frailey 2011）。本章我们简单介绍一下 4 种计算领域：信息安全、人工智能、云计算、大数据。对每个计算领域，我们重点关注 4 个方面的因素：

- 涉及哪些人；
- 关注什么问题；
- 涉及哪些计算基本原理；
- 如何为计算和所在领域带来新的基本原理。

这种分析有可能揭示一些新的基本原理，并帮助实践者理解计算能够给他们带来的利益和限制，也有可能帮助探索不同技术之间的联系，从而为未来的创新埋下伏笔。

在深入这些计算领域之前，我们应该进一步加深对计算领域与计算基本原理两者之间关系的理解。这种理解能够帮助我们更好地分析这些领域。

领域和基本原理

有两种基本策略来表示特定学科的知识体系。一种策略罗列出该学科包含的所有领域，另一种策略则罗列出所有的基本原理。对同一知识空间的不同表示会对实践活动产生不同的指导意义。在本章中，我们使用“领域”(domain）来表示技术领域，即关注特定技

术的领域。 21

教育者通常使用“知识体系”（BOK）一词来表示对特定学科知识的系统化描述，并且基于知识体系来设计相应的课程体系。ACM 在 1968 年给出了关于计算的第一个知识体系，并在 1989 年、2001 年和 2013 年进行了更新。1989 年版本包含了计算的 9 个核心领域（Denning et al. 1989），2001 版本包含了 14 个（ACM Education Board 2001），2013 版本则进一步扩充到 18 个（ACM Education Board 2013）。之所以将这些子领域称为核心领域，是因为这些领域都或多或少为其他领域提供了基础技术支持。

基本原理框架（如本书所给出的计算基本原理框架）与面向应用领域的框架是正交的。一条基本原理可能会出现在多个领域中，而一个领域可能会依赖于多条基本原理。这些被领域所依赖的基本原理，其演化速度远低于技术的演化速度。

虽然这两种风格的框架具有很大的差异性，但它们也存在紧密的关联。为了更形象地感受到这种紧密关联，我们可以想象这样一个二维矩阵：每一行代表一个领域，每一列代表一类基本原理，所有的单元格则代表了特定方面的知识空间（见图 2.2）。

图 2.2 关于计算的知识空间可以被表示为一个矩阵：其中，列代表不同类型的基本原理，行代表不同的领域。图中灰色背景的单元格中给出了信息安全领域使用到的两个协作类型的基本原理：密钥分配协议（用于安全地分配密钥）；零知识证明（用于在两个参与者之间安全地交换私密信息）

基于这种矩阵，我们可以说：面向技术的知识体系是对该矩阵中行的罗列，而面向基本原理的知识体系则是对矩阵列的罗列。这两种知识体系从不同的角度对相同的知识进

行了阐述。

设想一个人试图罗列出一个技术涉及的所有基本原理。这个人可以从基本原理的6种类别出发分析出该技术领域涉及的所有基本原理，即对应于矩阵中的一行（见图2.3）。在本章的余下部分，我们将用这种方式对4个领域涉及的基本原理进行分析。

	通信	计算	协作	记忆	评估	设计
信息安全	保密认证 隐蔽通道	加密算法的 复杂性	密钥分配协议 零知识证明	约束划分 引用监视器	协议性能分析	端到端层次化 功能 虚拟机

图2.3 安全领域涉及的计算基本原理在矩阵中对应于安全领域所在的那一行。如同大多数的计算领域一样，安全领域涉及的计算基本原理也可以划分为六种类型

这种基本原理框架也指出了另外一种分析方法：一个人可以罗列出涉及特定基本原理的所有技术（见图2.4）。

	通信	计算	协作	记忆	评估	设计
架构			硬件握手协议			
互联网			TCP/IP 通信协议			
信息安全			密钥分配协议 零知识证明			
虚拟内存			缺页中断			
数据库			锁协议			
编程语言			信号量监视器			

图2.4 协作技术表现为矩阵中协作所在的那一列。在几乎所有的计算领域中（包括列出的6种计算领域）都会涉及协作相关的基本原理

信息安全

信息安全是计算机科学中一个具有悠久历史的领域。甚至在任务批处理大行其道的时代，用户就开始关注存放于计算机中的数据的安全问题。计算机是否存放在一个物理安全的地方？在一个新的任务被装载前，存储器是否已被清空？操作者的错误或硬件失效是否会导致数据的丢失？

1960 年左右，随着第一代多道程序分时系统的出现，操作系统的设计者就已经对信息保护问题有了深刻的实践认知。他们设计出各种方式对主存储器进行分区，从而保证不同程序的代码和数据不会混淆在一起。这其中，虚拟存储器是最复杂的一种机制。他们发明了分层文件系统，使得用户可以真正地管理他们的文件，并设定其他用户对这些文件的访问权限。他们发明了密码系统以避免未经授权的用户使用计算机。他们构造了多种结构，以避免特洛伊木马或其他恶意软件对系统和数据的干扰和破坏。他们发明了多种访问控制机制，使得机密信息无法写入公共文件中。他们创建了针对计算机操作者的各种操作策略，以保护数据和防范入侵。自 1970 年以来，人们已经广泛认识到，保证信息的安全是操作系统设计者一项责无旁贷的任务（Denning 1971，Saltzer and Schroeder 1975）。

20 世纪 70 年代的 ARPANET、80 年代的 Internet、90 年代的万维网（World Wide Web）提供了一种世界范围的信息共享网络，也使得信息的丢失或偷盗成为一个更加现实的问题。密码系统在信息安全和身份验证中承担了核心角色（Denning 1982）。设计者在实践中遇到并应对各种问题，比如：数据库记录保护、密码保护、基于生物信息的用户验证、反入侵保护、入侵检测、病毒 / 蠕虫 / 恶意软件的防范、多层安全系统、信息流管理、匿名事务、信任分级准则、数据恢复等。法律实践专家开始遇到越来越多的针对计算机系统的犯罪行为，并开始唤醒每个人对个人信息重要性的认识。然而，很少有人认真对待这个问题。为了能够更快地交付新的系统，很多开发者开始降低对信息安全的重视程度。他们认为，等到信息安全问题真正出现后再亡羊补牢，也未为迟也。这种观点注定是错误的。大量可被轻易入侵的系统出现在重要的业务活动中，这些系统缺乏对信息保护的系统性考虑，因而对各种（善意、恶意或无意的）操作采取了宽松容忍的策略。

随着越来越多的金融数据、人事数据、个人隐私数据以及公司数据被存储在可在线访问的计算机中，对这些数据的恶意入侵行为出现了极大地增加。安全专家已经开始担忧信息战争的可能性（Denning 1998）。隐私专家则向公众奋力疾呼保护个人隐私数据的重要性，这关系到我们每个人的基本自由（Garfinkel 2001）。在 1999 年，公众开始普遍担心“千年虫”问题有可能导致网络的崩溃，这个问题产生的原因是在计算机中一个年份数

据被存储为两位而不是四位十进制数。自此以后，人们开始关注到信息网络的脆弱性问题以及保证信息安全的重大挑战。多国专家开始忧虑大规模的信息攻击行为有可能导致世界经济甚至人类文明的毁灭（Schneier 2004，2008，Clark 2012）。

表 2.1 给出了信息安全领域涉及的人、问题以及计算基本原理。

表 2.1 安全领域

人	角色	操作系统设计者，网络工程师，网络操作者，防御者，执法者，取证调查者，国土安全人员，公共政策官员，外交官，隐私权拥护者
内容	故障，问题，关注点	控制共享，存储保护，文件保护，访问控制，信息流，可信系统，加密通信，身份验证，签名，秘钥分配，基于数据关联的抗干扰
计算基本原理	通信	密码系统，保密，身份验证
	计算	单项函数，加密复杂性，哈希算法，形式化验证
	记忆	访问控制，错误限制，信息流，多级安全存储，引用监视器
	协作	秘钥分配，零知识证明，身份认证协议，签名协议
	评估	性能和吞吐量协议，安全系统准则
	设计	开放设计，最小特权，缺省故障安全，端到端设计，层次化功能，虚拟机
来源于其他领域的基本原理		信息安全实践，入侵检测，生物特征，取证规则，基于统计数据库的推理

人工智能

让机器可以执行人类智力活动的想法可以追溯到 5 个世纪之前。1642 年，Blaise Pascal 建造了一台机械式计算器。1823 年，Charles Babbage 发明了差分机，用于自动进行算数函数的计算。在 19 世纪后期，一个名为“mechanical Turk”的自动下棋机器人，在受到广泛关注后最终被发现只是一个恶作剧（Standage 2003）。实际上，很多这些类似的想法都可以被看作人工智能的萌芽（Russell and Norvig 2010）。

24
~
25

1956 年，在 Claude Shannon 和 Nathaniel Rochester 的帮助下，John McCarthy 在德国达特茅斯组织了一次研讨会。这次研讨会标志着人工智能领域的诞生。当时，研究者的一个基本观点是“原则上，人类智能的机理可以被精确地描述出来，因此，可以用机器来仿真人类智能”。这种观点是非常合理的，因为很多人类智力活动似乎就是在执行某种算法，而且人的大脑似乎就是一个可以执行算法的电子化网络。Herbert Simon 预测，到 1967 年计算机可能成为世界象棋冠军，可能发现并证明一些重要的数学定理，而且很多心理学理论可以体现在计算机程序中。他的第一个设想比预计时间迟到了 30 年才得以实

现，而剩下的两个设想目前仍然没有实现。 26

图灵（1950）为现代人工智能留下了很多火种：图灵测试、机器学习以及通过“成长”形成一台智能机器。图灵意识到，由于对“智能”一词人们始终缺少一个足够明确的定义，以至于无法有效衡量一台机器是否有智能或具有何种程度的智能。他提出的模拟游戏（即图灵测试）不再关注一台机器是否有智能，转而关注一台机器是否表现出智能的行为。他预测，到2000年，机器至少能够让70%的人类评审者在5分钟的时间内无法判断与之交谈的是一台机器还是一个人类。这个设想到目前还没有实现。

图灵关于智能的行为观点在人工智能领域的形成中起到了重要作用。然而，到20世纪70年代，这种观点受到了尖锐的批判。很多人工智能项目开始以设计“专家系统”为目标，即：构造出与某些领域（如医学诊断领域）的人类专家具有相同能力的智能系统。Hubert Dreyfus（1972，1992）则坚持认为，专家的行为绝不是基于规则的机器所能完全模拟的。当时，人们对这种观点嗤之以鼻，但时间证明了这种观点的正确性。只有非常少的专家系统表现出一定的实用价值，但没有一个专家系统能够达到人类专家的水平。John Searle（1984）则认为传统机器具有智能是不可能的：一个基于规则的机器可能可以使用中文与人对话，但机器根本不知道这些中文的含义。他不认同“强人工智能”的概念（即机器行为能产生意识），而更偏向于“弱人工智能”（即机器能够模仿出人类的行为，但这种模仿背后的机制可能与人类大脑的工作机制没有任何相似之处）。Terry Winograd和Fernando Flores（1987）认为人工智能基于某些人类的哲学假设，而这些假设可能根本无法解释智能的工作机理或根本无法导致智能的产生。

到20世纪80年代中期，人们逐渐意识到关于人工智能的很多初始设想很难在短时间内实现。提供研究基金的机构开始停止向人工智能领域提供新的资金支持并且要求已有研究项目提供更为确实的研究成果。缺少资金的支持，很多研究者无法展开具体的研究工作，转而开始认真反思这个领域面临的问题。人工智能领域的先驱者Raj Reddy将这一黑暗时期称为“人工智能的冬天”。

这一时期的反思酝酿了人工智能研究的一个新方向。人们不再关注如何对人类意识活动进行建模，转而去寻求构造一些能够替代人类认知活动的智能系统。自动认知系统的工作原理不需要与人类意识活动的原理相一致。它们甚至根本不去刻意模仿人类解决问题的过程。这一新方向更强调通过实验来确认所提出的自动化工作原理是否有用、可靠和安全（Russell and Norvig 2010，Nilsson 2010）。这一新方向最近引起公众广泛关注的进展包括：1997年IBM深蓝国际象棋程序击败国际象棋世界冠军Garry Kasparov，2010年 27

Google 的无人驾驶汽车，2011 年 IBM 的 Watson 计算机在益智问答游戏节目“危险边缘”中获得冠军（击败人类对手）。这些计算机程序中使用到的方法具有极高的效率，但却没有刻意模仿人类思考或大脑运作的机制。同时，这些方法仅对解决特定的问题有效，而不具有通用性。

计算机科学、认知科学、医学及心理学领域的很多研究者仍然在研究人脑的工作机理和意识的产生机理。Kurzweil 的畅销书《奇点临近》（Kurzweil 2005）和 2013 年对外公布的研究项目 Brain Activity Map Project 表明，这个研究方向仍然具有非常强大的吸引力。

人工智能的重生显示了巨大的成功，进而产生了一些崭新的关注点。在与机器的竞赛过程中，Erik Brynjolfsson 和 Andrew McAfee（2012）发现，自动化的浪潮正逐渐取代人类在知识相关领域的工作，正如在上个世纪中机械自动化取代了大量的人类体力劳动。知识自动化的实例包括：电话转接中心、语音菜单系统、在线商品交易、在线银行、政府服务、出版、新闻传播、音乐发布、广告、监管、反恐等方面。笔者所担忧的是我们正滑向一个不需要大量人工劳动的社会，而这个社会又无法为大量失业的人类个体提供充足的资源。

表 2.2 给出了人工智能领域涉及的人、问题以及计算基本原理。

表 2.2 人工智能领域

人	成员	AI 专家，AI 实践者，规划师，下棋者，贝叶斯学习者，机器学习者，仿生设计师，认知科学家，心理学家，经济学家，执法者，机器人专家
问题	故障，问题，关注点	认知任务的自动化，经验规则的设计与试验评估，演化计算，遗传计算，神经计算，模式识别，自动分类，语音识别，自然语言翻译，人造脑，超人类智能，自治系统（无人驾驶飞机、汽车等）
计算基本原理	通信	噪音信道模型
	计算	经验算法，分类，贝叶斯推理，机器学习，大状态空间搜索，智能的模型
	记忆	存储模型，稀疏分布式存储，神经网络检索，局部学习算法
	协作	训练协议，协调理论
	评估	经验评估的实验方法；查全率，查准率，准确性
	设计	监督实验和无监督实验中大数据集的存储
来源于其他领域的基本原理		意识的产生机理，言语行为理论，语言学，神经科学，统计推断

云计算

云计算是一个现代的时髦概念，其背后隐藏了关于信息共享和分布式计算的丰富含

义。大量计算设备的互联产生了一种规模经济：在其中，人们不再关注服务和数据存放的物理位置。“云”这个术语出现在 20 世纪 90 年代后期，可能是在相关的技术或市场研讨中，人们使用云来比喻由计算机构成的网络。

早在 20 世纪 60 年代中期，MIT 的 MAC 项目已经体现出了类似的想法，即：构建一种能够向众多用户提供计算资源共享的系统。MAC 是词组“ Multiple-Access Computer”中单词的首字母缩写，有时也代表“ Man And Computer”。该项目产生了 Multics 系统，一个强大的可以对内存、外存、CPU 等昂贵的计算资源进行分时复用的多道程序系统（每个用户使用这些资源的成本得到了极大的降低）。J. C. R. Licklider 是这种思想灵感的提出者。他认为，计算资源应该成为一种基础设施，任何人都应该可以很方便地接入其中（Licklider 1960 ）。

在 1969 年底开始运作的计算机网络 ARPANET 使得“计算成为基础设施”的宏伟设想得以成真。ARPANET 的设计目标就是资源共享：接入网络的任何用户都可以与网络中的其他计算节点互联并使用其上提供的服务。因此，没有必要去复制一个已经共享的服务（在服务质量能够得到保障的前提下）。ARPANET 的设计者很快意识到，共享服务应该通过名称而不是物理位置来进行访问，因此，开发一种位置无关的寻址系统对于屏蔽网络通信的底层细节至关重要。Vint Cerf 和 Bob Kahn 发明了 TCP/IP 通信协议（ 1974 ），能够支持互联网上的任何两台计算机在仅知晓对方 IP 地址（一种逻辑地址）而无需知晓物理地址的情况下进行信息传递。1983 年，ARPANET 开始正式使用 TCP/IP 协议。

1984 年，ARPANET 开始使用域名系统，一种将特定的字符串映射到 IP 地址的在线数据库（例如，将字符串“ gmu.edu”映射为 IP 地址“129.174.1.38”）。域名系统带来了另一个层次上的位置无关性：用户现在只需要记住一些具有语音信息的字符串，就能够访问对应的互联网服务。

28
~
29

到 20 世纪 90 年代，万维网（ World Wide Web）的出现使得任何信息对象都可以在互联网上进行共享。信息对象使用 URL（Uniform Resource Locator，统一资源定位符）进行命名。URL 的基本格式是“主机名 / 路径名”，其中路径名指出了一个信息对象在给定主机上的目录位置。20 世纪 90 年代中期，Robert Kahn 设计了一个互联网服务 handle.net。该服务可以把一个具有唯一性的标识符映射为一个信息对象的 URL。基于此，他还为美国国会图书馆和大多数的出版商设计了 DOI（ Digital Object Identifier，数字对象标识符）系统（Kahn and Wilensky 2006 ）。DOI 提供了更高层级的透明性：一旦为一个数字对象赋予一个 DOI 之后，这个 DOI 将永久指向该数字对象，无论该对象存放在何处或何时被创

建（Denning and Kahn 2010）。

面向终端用户的分布式计算服务的技术架构在近年来持续发展。Multics 系统使得一个大型计算机系统可以在其用户间分享计算资源。20 世纪 70 年代，施乐 Palo Alto 研究中心（PARC）构造了 Alto 系统，一个由连接至以太网的一组图形工作站形成的网络（Metcalfe and Boggs 1983）。施乐将该系统的体系结构命名为“客户端 – 服务器”架构，因为用户总是通过一个本地的接口（客户端）来访问网络上的服务。X-Window 系统（1984 年源于 MIT）是一个通用的客户端 – 服务器系统：它支持一个新的服务提供者将其硬件和用户接口接入网络，而无需设计相应的客户端 – 服务器通信协议。今天，大多数的 web 服务都使用了客户端 – 服务器架构：服务提供者通过展现在标准 web 浏览器上的界面为用户提供服务。大多数云中的服务也采用客户端 – 服务器架构，其中 URL 系统对服务器的物理位置进行了完全的屏蔽。

表 2.3 给出了云计算领域涉及的人、问题以及计算基本原理。

表 2.3 云计算领域

人	角色	网络设计者，分布式计算设计者，客户端 – 服务器架构师，企业系统设计者，商业人员，政府人员，经济学家
内容	故障，问题，关注点	位置无关的服务与数据存储，备份，分布式计算模型
计算基本原理	通信	纠错代码，声音与图像文件的压缩，基于位置的识别
	计算	Map-Reduce 方法，大规模并行计算
	记忆	数据复制，原子事物，事务回滚，数据库结构，互联网搜索，命名，数字对象标识符，数字对象句柄
	协作	锁协议，回滚协议，文件传输协议，文件同步协议，超文本协议，域名系统，时间戳，版本控制
	评估	大规模并行和分布式存储系统的性能
	设计	接口设计，数据仓库体系结构
来源于其他领域的基本原理		社交网路，电子商务，关键基础设施建模，位置统计推断

大数据

大数据是最近出现的另一个时髦概念，其背后隐藏了关于计算的丰富信息。大数据关注如何对互联网上的海量数据进行分析，从中发现有价值的统计规律和相关性等信息。这种分析可以广泛应用于各种领域，例如科学、工程、商业、人口普查、执法等。

计算机科学家对数据的存储、查询及处理已经进行了长时间的关注，而且很多关注的问题甚至比目前的技术进展还要超前。可惜的是，这些超前的想法由于各种因素的影响被埋没在历史的尘埃中，被大众所遗忘。“大数据”这一术语在很大程度上是新瓶装旧酒，

虽然这一术语确实对很多领域产生了显著的影响。例如，在商业活动中，商业组织收集海量的客户相关数据，并利用这些数据去发现市场趋势、广告投放对象以及客户忠诚度等信息。受到公共资金资助的科研项目也被要求对外公开其数据，以方便公众和其他科研项目能够对这些数据进行多方面的利用和分析。警察系统则利用海量的通信信息和信用卡交易信息，从中发现犯罪分子。所有这些领域都开始主动寻求数据科学家、数据分析师以及数据系统设计师来帮助他们进行数据分析工作。

计算机科学家在其中的贡献主要体现在两个方面。一方面是关于更高效地数据分析方法，另一方面则是能够支持海量数据处理的系统或技术架构。例如，Richard Karp（1993）基于组合方法实现了对基因数据片段进行融合从而形成基因组图谱的高效算法。Tony Chan 和 Yousef Saad（1986）的研究工作表明，hypercube（一种早期出现的并行计算架构）对于多重网格算法（一类重要的数字计算方法）具有最优的效果，而多重网格算法能够对大规模数据空间的数学模型进行求解。Jeffrey Dean 和 Sanjay Ghemawat（2008）设计了 MapReduce 算法，能够支持数千个处理器通过并行的方式对海量数据进行处理。

30
2
31

在商业领域中，如何对大规模数据集进行处理和分析一直以来都是一个重要的问题。商业组织会收集关于客户、库存、产品制造、财务等方面的各种数据，这些数据对于一个大型的国际化商业组织的正常运转具有非常重要的作用。20 世纪 30 年代，一个电子计算机还未出现的年代，IBM 靠出售类似卡片分类器和检索器的简单设备从数据处理市场获得了巨大的财富。20 世纪 50 年代，IBM 开始向电子数据处理领域发展，转型成为一家计算机公司。1956 年，IBM 对外发布了第一个硬盘存储系统 RAMAC 305，受到了广泛关注。IBM 声称，任何商业组织都可以将其堆满仓库的文件资料转移到一个小小的硬盘中，进而能够对数据进行极为高效的处理。随着数据存储需求的不断增长，设计者开始关注如何对数据进行有效的组织从而实现对数据的快速访问和简易维护。当时，两个主流且存在竞争关系的方法分别是综合数据系统（Integrated Data System，IDS）（Bachman 1973）和关系数据库系统（Relational Database System，RDS）（Codd 1970，1990）。综合数据系统具有简单、快速、实用等特点，能够在管理大量数据文件的同时隐藏文件在硬盘上的物理结构和位置。关系数据库系统则基于数学化的集合理论，它具有一个非常清晰的概念模型，但在经过了多年的发展后才实现了与综合数据系统相当的处理效率。从 20 世纪 70 年代开始，研究领域形成了一个关于大规模数据库（very large databases）的研究团体，并每年召开一次学术会议（VLDB）对相关议题进行讨论。

从 20 世纪 50 年代开始，计算领域的研究者进入了文档管理领域：帮助文档管理员

组织数据以实现更加快速的文档检索。图书馆是这些信息检索系统的第一代用户。研究者开发了模糊查询系统。例如，用户可以发出“请查找关于信息检索的文档”，而返回的文档中不一定包含“信息检索”这个字符串。今天，互联网就是一个巨大的无结构的存储系统。在互联网上进行关键词检索非常快速但却不够准确，因此，有效的互联网信息检索仍然是一个困难的问题（Dreyfus 2001）。

Gartner Group 将现代的“大数据”定义为 4V：数据体量巨大（Volume）、数据的产生速度快（Velocity）、数据的表现格式丰富（Variety）、数据对决策活动具有重要的支持作用（Veracity is important to decisions）。从 2014 年开始，数据科学的课程或关于数据科学
32 的研究中心在大学和其他研究机构中如雨后春笋般出现。多个领域都涉及其中，例如，来自运筹学和统计学领域的分析师、来自计算机科学和信息系统领域的架构设计师以及来自建模和仿真领域的可视化工程师。这些实践和研究活动也确立了“数据科学”领域的主要研究问题：寻找对大规模数据集进行处理和分析的科学理论基础。

表 2.4 给出了大数据领域涉及的人、问题以及计算基本原理。

表 2.4 大数据领域

人	角色	商业人员，政府机构，企业设计者，科学数据收集者，统计学家，大系统建模者
问题	故障，问题，关注点	超大规模数据集中相关项的发现，计算复杂性，隐私问题，推断，数据恢复取证，信息检索
计算基本原理	通信	从大规模传感器集合向集中存储器的可靠信息传输，数据破坏、修改及丢失检测，数据控制权的非法转移检测
	计算	数据分析算法的计算复杂性
	记忆	存储，备份，错误控制，数据存在性测试，数据物理位置测试，超大规模数据库数据恢复取证
	协作	Map-Reduce 计算
	评估	超大规模网络中大规模查询和分析的完成时间预测
	设计	数据备份，数据索引，最优化检索的组织结构
来源于其他领域的基本原理		自然语言处理，统计推断，众包，情感推断

总结

本书采用的关于计算重要原理的框架提供了一种有效的方式去分析特定技术所涉及的基本原理。这种框架也可以用来分析特定计算应用领域背后所基于的计算基本原理，在这些领域中，具有不同技术或工程背景的人之间相互配合来解决该应用领域中存在的
33 问题。

第3章
Great Principles of Computing

信　息

通信的内容语义与通信工程无关。

——Claude E. Shannon

软件并不只是交互设备，更生成了一个用户生活空间。

——Terry Winograd

自从数学家、通信工程师克劳德·香农（Claude E. Shannon）在20世纪40年代发现信息论以来，关于信息的研究迅速发展。信息论中的一个关键准则就是信息和含义是有区别的，这就使得机器可以忽略含义而去处理信息本身。然而，整个通信和计算的目的是传达并产生有含义的结果，这又将如何实现呢？

软件设计师、科学家和消费者都希望软件能够生成虚拟世界、社交网络、音乐、新发现、财务预测、情书和令人激动的图片等，甚至更多。例如，一只股票的报价表在财务外行看来可能是数字乱码，而对于专业投资者来说却有着极大的价值。当所研究的基本对象具有一定主观性时，我们又该如何界定信息科学呢？

这些问题看起来有些自相矛盾，因为信息论认为含义和计算机器无关——这似乎和人们使用计算系统的经验是矛盾的。此外，在很多人看来信息的概念似乎又是模糊和抽象的，难以理解信息系统实际是如何工作的。这一章的目的就是说明信息是非常真实的，是以一种物理可见的模式存在的（见图3.1）。我们调研了信息论中必然会提到的有关表示和传输信息的概念、信息论如何与可计算性理论相结合、信息论的局限又在何处。经典的信息论无法解释信息的含义以及新信息的产生，而这个调研则说

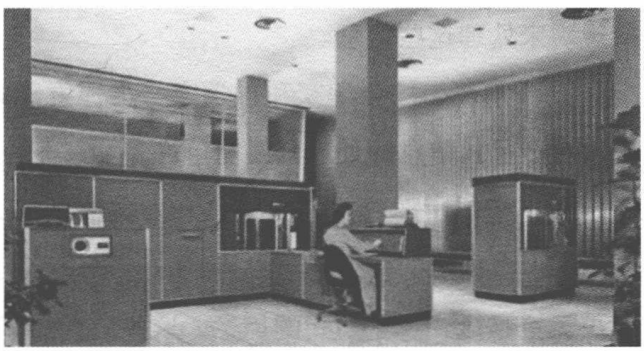


图3.1　1956年IBM发明了世界上第一套磁盘信息存储系统，RAMAC 350。这个机器的宣传片显示了疲惫的秘书们在文件柜走廊中穿行的情形。影片显示，RAMAC 350能够存储大约5立方米文件柜中的所有内容，并且能够支持几乎实时的数据搜索。该影片也显示出使看似抽象的信息概念具象化的尝试，这对于当时的观众已并不新奇（照片由IBM提供）

明了其中的原因。我们通过描述一个含义保留变换的模型来解决这个明显的悖论。

信息的表示

人类在通信时是非常灵活的。这里有四个例子，其中前两个例子阐明了什么是显式含义：

1）当我们指向某个物体并告诉朋友这个物体的含义时，这个物体就“携带”了信息，因为从现在开始我们的朋友无论何时看到这个物体，这个含义都会在大脑中触发。

2）当我们发现某些现象模式重复出现时，就会为这样的重现模式命名。当我们再次看到这样的模式出现时，便可以预测结果。因此，这个模式就携带了可预知结果的信息。科学的目的就是发现自然界中重现的现象，而工程的目的就是将这些重复的模式转变为可利用的技术。

36 接下来的两个例子说明了什么是隐式含义：

3）社会群体会定义一些重现方式来交流信息。例如，很多司机将要进入高速公路时，会一边缓慢地靠近高速通道一边开启方向灯，但并没有成文的规定要求他们这样做。

4）人们日常生活中的很多习惯和惯例是没有命名但携带信息的。例如，在大多数文化中，“过来”的手势传递的就是让别人靠近你的信息。

科学家和工程师的工作就是构建技术来处理显式的信息，也就是建立信息的物理表示与预期含义的关联，如使用电磁信号对人的声音进行编码。这样，我们通过声明表示与含义之间关联的方式来产生信息，然后通过在存储器中存储这些表示方式并且用不同的变换规则来处理这些信息。

千百年来，社会学家和哲学家努力探索隐式的信息，通常很少有一致的看法。而工程师对于显式信息的处理则要简单许多。

人工智能试图跨越显式信息和隐式信息的边界，工程师正在寻找既能识别隐式信息又易于人类理解的信息表示方式。

无论是显式还是隐式的信息，这些信息的存在都建立于人类认知的一致。我们理解某种表示的含义，因为要么有人直接告诉我们如何解释它，要么我们间接通过经验学习到。

计算机和通信工程师将信息编码成电磁信号进行传输。例如麦克风将人的声音转变为电信号，然后通过一个磁盘记录这个信号的副本，最后扩音器将这个磁盘中的信号转变为声波。无线发射机将声音信号叠加在射频信号中，通过射频的幅度来表达这个信号，而接收机只要减去原本的射频信号就可以提取出声音信号。工程师对于如何编码信息表示方式及其含义必须准确一致，否则这套物理系统就会出错。

计算机和通信工程师使用比特（二进制数字）作为信息的基本单位。香农在 20 世纪 40 年代中期引入了“比特”，那时是计算机时代的初期。尽管使用十进制来构造的硬件元件也可以使用，并且早期也有一些计算机使用这些硬件，然而采用二进制元件因更加可靠 37 而逐渐变成行业标准。香农发现二进制计算电路的功能可以用逻辑公式来表示，该公式中只包含“真”或“假”两种变量。因此，比特模式可以表示计算机电路。电路处理的数字就是这些二进制数字，也就是电路表示的数字本身（见图 3.2）。自 20 世纪 50 年代以来，计算机完全变成了二进制，无论是逻辑电路还是其数据存储。

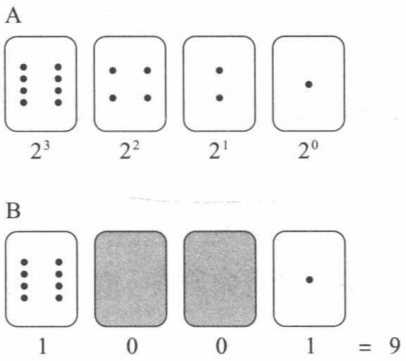


图 3.2　儿童使用卡片来快速学习二进制数字。上图：每张卡片都比其右边的卡片多一倍的点数。4 个儿童列成一排，引导他们通过举起卡片来表示不同的数字。下图：当第一个和第四个儿童举起卡片而第二、三个儿童藏起卡片时，数字 9 出现了。通过这种方式，儿童很容易掌握二进制。由于任何信号都可以数字化为二进制数字，任何文本文件也可以被编码成二进制数字，因此位成为了信息表示和量化的通用方式（由 Tim Bell 和 Mike ellows 提供，csunplugged.org/videos）

香农还证明了实际上大部分通信系统中的连续信号也可以数字化，而数字化引起的一些微不足道的误差完全可以忽略，稍后将简要说明。

所有的数据形式，包括数字、信号、逻辑公式、文本等，都可以表示成位模式，位成为衡量信息量的标准单位。现代词语中的“24 位颜色”“100MB 通信”“32 位电脑”和“256 位密钥”等都包含位的概念。在 20 世纪 60 年代，计算机制造商开始使用“字节”（即一组 8 位信息）来表示 ASCII 扩展码中的单个字母、数字或标点符号。后来，计算机处理的数据呈指数增长，于是人们开始使用新的希腊文前缀来命名这些数据（见表 3.1）。其中每一个前缀都表示前一项前缀的 1000 倍（或 1024 倍，2^{10}）。在 20 世纪 60 年代，磁盘和 38 内存容量通常用千字节来衡量，到了 80 年代，便用千兆字节来衡量，而那时的 NASA（美国国家航空航天局）却一直苦恼于如何存储每日卫星接收到的 1TB 的数据量。到了 2014 年，“大数据”用于描述 PB 级字节的数据量，同时每年因特网的数据量都超过了 1ZB 字

节。思科公司（2012）预测网络规模和数据将持续以指数形式增长。

表 3.1 数据单位名称

名称	十进制表示	二进制表示
byte (B)	8 比特	2^3 比特
kilobyte (KB)	10^3 字节	2^{10} 字节
megabyte (MB)	10^6 字节	2^{20} 字节
gigabyte (GB)	10^9 字节	2^{30} 字节
terabyte (TB)	10^{12} 字节	2^{40} 字节
petabyte	10^{15} 字节	2^{50} 字节
exabyte	10^{18} 字节	2^{60} 字节
zettabyte	10^{21} 字节	2^{70} 字节
yottabyte	10^{24} 字节	2^{80} 字节

通信系统

通信系统是最简单的信息系统，在 1948 年一篇名为《通信的数学理论》的文章中，香农提出了通信系统的第一个理论模型（Shannon 1948）（见图 3.3）。其本质是如下过程：消息源发送一条消息，编码器按照编码本规定为这条信息生成一个独有的信号；信道作为中间媒介从源到端运载这个信号；接收端的解码器使用同样的编码本将这个信号转换成初始形式——消息就到达接收处了。香农的模型适用于任何使用编码、解码、传输、存储或是查询信号数据的系统，甚至可以作为科学发现的模型：将自然界看作现象（消息）的源头，而传输媒介（即通道）就是探索的过程（Dretske 1981）。

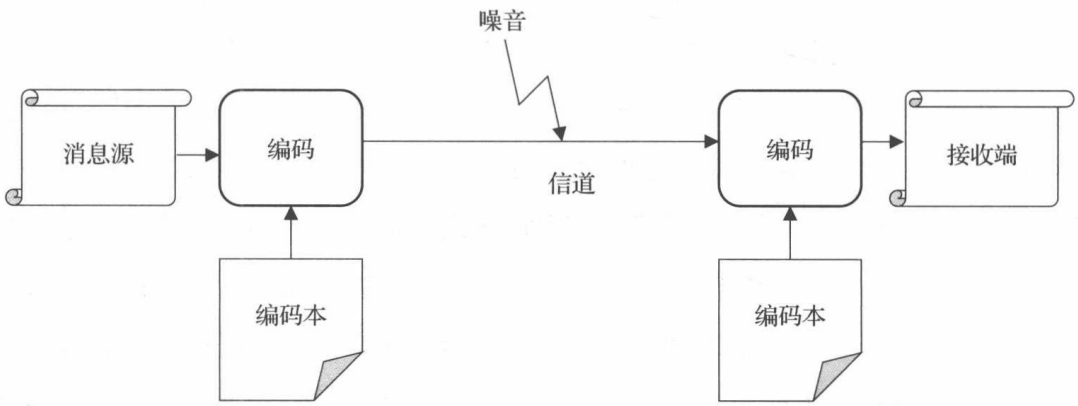


图 3.3 克劳德·香农（1916—2001）描述了一个信息系统模型，它是当今信息论的基石。消息源代表所有可能要发送的消息的集合，信道是存储和运载信号的媒介，编码是将消息转换为信号，解码是将信号还原为消息，编码本包含了消息与信号互相转换的规则，噪声是指任何改变信号的事物

噪声是通信模型中的一个重要元素。任何在传输通道中改变信号，从而导致解码出错误消息的干扰都是噪声。通信技术中噪声的例子比比皆是：雾气和黑暗干扰了船只之 39
间的信号通信；电报站之间过长的距离减弱了信号的强度；雷电干扰了调频广播的传输；DVD 上的划痕会导致读取失败；环境的声音淹没了演讲的声音。

值得注意的是，在通信系统中，编码和加密是不一样的。加密是一个额外的步骤，在消息到达编码器之前将消息转换成密文，这样只有拥有密钥的接收器才能够读取消息。在这种情况下，通信系统的工作就是将密文准确地传输给接收方，如果接收方具有密钥则可以进行解密。

正如前文所提到的，香农将他的数学模型在二进制位上进行标准化，他认为所有的信号都能够用二进制位表示，这一过程在字面上被称为数字化，即将模拟信号转变为数字信号。数字化并不是生成信息的一个完全拷贝，而是一种近似化过程，因为其常常会丢失一些信息。显而易见的例子有许多，如物体边缘参差不齐的像素化照片，同时也有一些情形是非常微妙不那么直观的。物理现象中的定量分析，比如火星探测器的轨道位置，实际上不能通过计算机的有限运算进行精确表达。舍入误差会随着计算步骤而逐渐累加，导致整个计算 40
的精确度出现问题，使火星着陆器面临危险。更糟糕的是，一些计算步骤会放大错误。如两个几乎相等的数可以认为它们的差近似为零，在用这个近似的差除另一个数时会导致严重错误。数学软件的设计师设计了很多巧妙的技术，来防止这些数字化错误破坏计算结果。

Harry Nyquist 可谓当代的香农，他指出了上述普遍规律中的一个重要例外：通信系统可以免受数字化错误的影响（Nyquist 1928）。每一个连续、带宽有限的信号都可以无损地数字化，只要用至于两倍于最高频率的采样率进行采样。例如音频光盘（CD），为了使得质量没有明显损失，要每秒记录 44 100（44.1 千赫）个采样点，因为几乎没人能听到高于 22 000 赫兹的声音。

香农认为，由于我们可以对任何信号进行数字采样，同时也由于真实的通信系统具有有限的带宽，那么将通信模型限制为二进制序列，不会有任何损失。这不但简化了数学运算，而且使得这一结论适用于所有实际信道。

作为一个实例，一段简单的编码就可以说明通信模型的各种特性。假设一个消息源只传输消息的 1/4，我们把完整的消息定为 A、B、C、D，为这些字母分配 2 位的编码：

A：11

B：10

C：01

D：00

只能表示四种消息的编码在自然界中并不少见，我们细胞中的DNA序列就是一种只使用四个字母的自然消息源——G、A、T和C，它们是DNA中四种核苷酸的首字母。

如果消息源想传输序列“CAB”，那么就在信道中传输“011110”这个位序列，接收端会逆向这个过程，即在编码本中查找每一对位码然后输出对应的字母。

在任何关于编码的讨论中，我们都需要在编码长度（码字的位长）和信道抗扰能力间
41 做出权衡。短的编码更高效、传输更快并且占用较小的存储空间，然而短编码非常容易受到噪声的干扰，信道中仅仅一位的改变就会将码字完全改变。例如，如果信道将A的第一个位编码变为0，接收端就会收到01，然后解码成C而不是A。其中一个解决方案就是使用奇偶校验位来提示接收端是否收到错误信息，当原编码中有偶数个1时，奇偶校验位为偶数，反之则为奇数。下述编码就是在原始编码中添加第3位为奇偶校验：

A：110

B：101

C：011

D：000

现在信道噪声将A的第一位编码变成0后，接收端收到010时，会发现编码错误，因为010这个编码是无效编码。概括来说，1位的错误会导致1的数量变为奇数，从而标识这是一个未编码的模式。

然而，单一的奇偶校验位并不能标识出哪一位被改变了。上例中，接收端知道010是一个无效的编码，但是并不知道这三个编码（A、C或者D）中的哪一个被这个位错误改变了。通过添加冗余的编码，解码器不仅可以检测是否有错误编码，还可以定位具体受影响的消息位。试想下面的例子，在原来编码的基础上添加了3个位：

A：11111

B：10010

C：01001

D：00100

这个编码满足如下准则：加上额外的编码位，每一个码字都与别的所有码字至少有3位不同。若有1位发生错误，接收端收到编码后会发现这个错误编码与正确编码只有1位不同，但是与其他的字母编码却有2位甚至更多的不同。这样解码器就可以检测并修正只有1位发生错误的编码情况。

通信工程师理查德·海明（Richard Hamming）在1950年首先提出码字之间的距离

计算方式。两个码字之间的距离就是不同位码的个数，这就是后来有名的“海明距离”。海明指出，若要纠正 k 个字符，编码就必须包含足够的位使得码字之间的最小距离至少为 $2k+1$。他也发明了一系列编码，也就是现在的海明码。海明码在 2^k-1 位的码字中 42
嵌入 k 个校验码，然后使用一种简单的方法来构造电路用以将受损的比特位转换为原来的编码值。最有名的一种海明编码是（7，4）编码，在 4 个数据位中嵌入 3 个校验位，构成 7 位的编码。在计算机处理器和存储器之间传输数据时，海明码被广泛用于纠错。

当噪声在码位上随机分布时，海明码能够正常工作。然而，在某些信号中噪声可能会爆发性地出现，比如日晕会影响某些深空信号数秒，光碟上的划痕会损坏一系列邻近的凹点，这些噪声被称为*突发错误*。另一种类型的编码——Reed-Solomon 码便是用于检测和消除突发错误的。它的数学计算更加复杂，但是也和海明码一样很容易用高速数字电路实现。

不同于信号，位并没有实际物理意义。1 位可以表示可察的任何两种属性之一。比如，工程师可能让“1”代表激光束在 DVD 表面某点的反射，而“0”则代表没有反射；或者“1”代表晶体管的输出是 5V，而“0”则可能代表输出是 3V；或者“1”代表一特定频率（比如 400Hz）出现在一段音乐录音中，而“0”则代表不是该频率。位是一种抽象表达，用来声明我们想让系统做什么。工程师将这些物理的“东西”（材料）进行组装，用以完成所定义的功能。

物理系统中的信息总是要由物理状态来表达，而读写和变换这些信息也需要花费时间和精力，因此通信和计算从来都逃脱不了物理世界的限制。计算机芯片工程师知道蓄热和尺寸大小（芯片上所有电路元件的平均尺寸）等是影响他们能把电路做成多小的实际限制所在。同时每一个操作的时间消耗量则限制了可用时间内可计算的指令数。尽管新的算法在常规难题优化上有极大的改善，但计算量极大的一些问题仍然非常棘手，因为其物理运算所需要的时间大大超出我们的可等待范围。例如，在目前广泛应用的 RSA 加密系统中，为了寻找构成 600 位密钥的两个因子，即使利用目前最快的计算机也需要几百年的时间。

这些年，存储和计算能力呈指数增长。在香农发表其文章的同一年，在同一个地方——贝尔实验室，电子计算机就开始使用新发明的晶体管来取代真空管。电路设计师 43
能够压缩晶体管的大小，每 18 ~ 24 个月的时间就能够无额外费用地将接近两倍的晶体管放在同样大小的物理空间。这种压缩体积的过程已持续了 50 年，使得每 10 年就有了 100 倍计算能力的增加，这个趋势就是广为人知的摩尔定律。英特尔的创始人之一——戈登·摩尔（Gordon Moore）在他 1965 年的论文中首次描述了摩尔定律（Moore 1965）。

摩尔定律带给人们两种效应和影响。一个是惊人的计算能力，这对于20世纪40年代的计算科学先驱们来说如魔法一般；另一个是正如James Gleick（2011）所称的如洪水般的信息。第一种影响——每一个传输和存储信息的更小物理机制，都会导致第二种影响——信息的极大丰富。人们被巨量的信息淹没，无法消化信息，变得不知所措。

庞大的计算能力导致了一个流行的错觉，那就是计算机所操作的是位而非原子，因此计算结构在尺寸和能量上没有物理限制（Negroponte 1996）。从物理的角度来看，这个概念是完全错误的。因为抽象的位必须依靠物理介质来记录，而且机器要通过介质来获取位信息。这个记录的过程将我们带回了原子世界：没有它们我们无法完成计算。我们可以在极小的物理元件上进行快速的计算、传输和存储，但从来无法消除它们的时间和功耗。

信息的测量

香农设计了一个用于测量信息源中所含信息的方法，因为他想知道信息源中最短码的长度。编码的位数量并不能作为一个衡量标准，正如我们在上文中所看到的，单个信息源可以用任意数量的不同编码来表示。他的结论是一个好的衡量标准应该是消息集合中最短编码的长度，若编码再短一些则无法传输消息源的所有信息。

他认为衡量信息不应该依赖人类所观察到的编码的含义，而应该忽略信息的上下文，寻求编码、传输和解码的固定工作机制。邮政服务遵循相似的准则：他们的分发和传输系统从不依赖于所运输的信封中的内容。香农非凡的洞察力表现在“将信息的接收等同于不
44 确定性的减少”。他定义信息为判断信息源正在传送哪个消息所需要回答的是非（yes-no）问题的最小数量。我们越了解某个信息源可能会发送什么消息，那么看到这条消息时所获得的信息量就越少。

假设你知道某人将会只回答一个是非问题，但提前无法得知回答者的答案，那么回答者通过回答解决了你的不确定性。香农认为回答者只给了你一位的信息（1或0），即从两种可能答案中选择其中一个。当答案有两个以上时，需要更多的位来区分发送的消息。

假设我们想在电话簿中找到含有某个朋友名字的那一页，那么需要多少位来对页数进行编码呢？一个聪明的方法回答了这个问题：我们从中间打开电话簿，然后询问哪一半含有朋友的名字（由于使用字母顺序编写电话簿，这个问题就很容易回答）。然后我们把含有名字的那一半电话簿再分割成两半，询问同样的问题。重复这个步骤，直到只剩一页，这个朋友的名字就应该在那一页上。这个重复的问题（“哪一半”或者说“是否在左

边一半”）让我们快速定位。对于一本有 512 页的电话簿，第一个问题留下了 256 页来搜索，第二个问题留下 128 页，然后 64，32，16，8，4，2，最后是 1。需要 9 个“哪一半”问题来寻找包含名字的那一页。因此，当找到包含朋友名字的那一页时，我们获取了 9 位的信息量。

在构建编码时，人们需要思考消息的发生概率。塞缪尔·莫斯（Samuel Morse）发明了莫斯密码，并将之应用于 19 世纪 30 年代他参与发明的电报中。他分配最短的编码（单个点）来表示字母 e，因为 e 是在英文中使用最频繁的字母（大约 12%）。他将最长的编码分配给字母 j，因为 j 是最少使用的字母之一（约 0.15%）。这样的分配减少了传输的平均长度。图 3.4 说明了用以识别一个消息的问题可以用来定义信息编码，以及关于消息发生概率的先验知识会减少编码量。

假设我们有一长度为 L_i、概率为 P_i 的码字集合。那么编码的平均长度就是

$$L = \sum_i L_i P_i$$

对于图 3.4 中的编码，公式计算出第一种编码的平均长度是 2 位，而第二种编码的平均长度是 1.75 位。

45

在最优编码中，码字的长度即最小的 L 是多少呢？香农在他 1948 年论文的附录中回答了这个问题。他表示最优码字的长度是以 2 为底的码字出现概率的对数的相反数，即 $-\log_2 P_i$。因此，最优编码的平均长度是

$$H = -\sum_i P_i \cdot \log_2 P_i$$

这个公式和热力学的熵公式具有同样的形式，也有着相似的解释。熵是衡量系统状态的混乱程度或是不确定性的指标。一个系统的状态越混乱，熵就越高。最大的混乱度发生在所有的状态都有同等发生概率的情况下，最大的有序度则发生在某一种状态非常确定而其他的状态都绝不会发生的情况下。

香农认为熵是信息源中固有信息的衡量标准。一个信息源由一组可能的消息和消息发生的概率构成。熵只取决于消息的概率而不是它们的编码，熵可以确定最短可能编码的平均长度。任何更短的编码都将导致混淆从而无法得到唯一的解码结果。如下面的例子：

A：1

B：0

C：01

D：10

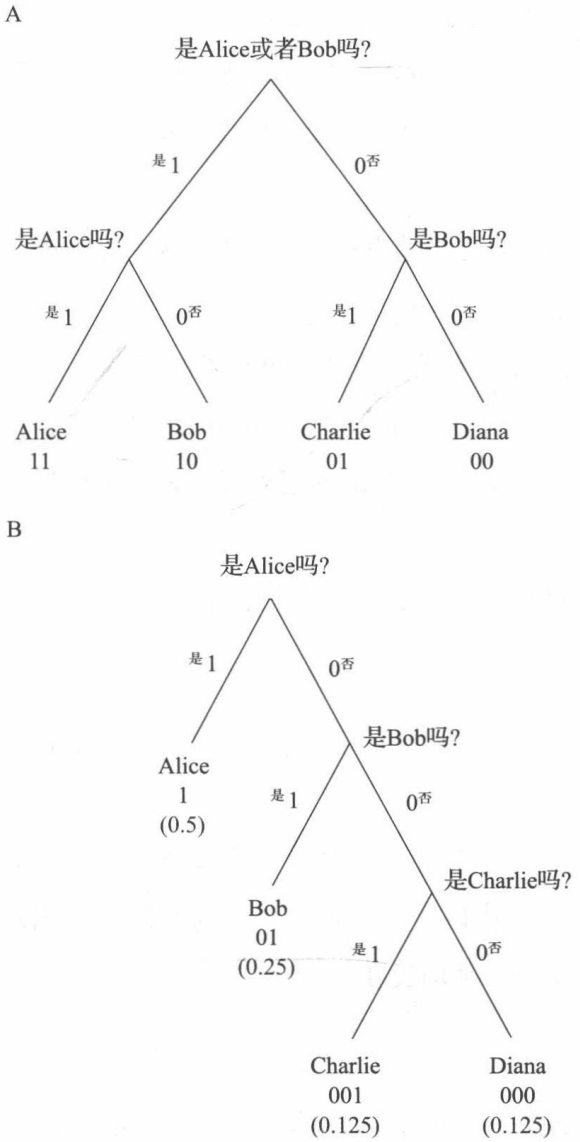


图 3.4 香农将一个消息中的信息量定义为将该消息从消息源中筛选出来需要回答的是非问题数，这些问题是用来减少“哪个是被发送消息”的不确定性的。试想，如果我们要从四个人中发现被某任务选中的人，则可以使用一个简单的决策树（上图），我们问：“是 Alice 或者 Bob 吗?”如果答案为是，则决策选择左边的子树。再问一个问题“是 Alice 吗?”，答案就揭晓了。每个个体的编码就是指向他 / 她的是非问题答案的路径。如果知道某个个体被选中的概率（下图括号里的数字），那么我们可以构造一个等级决策树，从而使得编码长度不等。例如，如果 Alice 是最有可能被选中的，我们就把编码 1 分配给他。Bob 是下一个最有可能的，则分配编码为 01，然后是 Charlie 和 Diana，他们有相同的选中概率，则都被编码为 3 位

如果以上这些消息出现的概率分别是 0.5、0.25、0.125 和 0.125，平均编码长度就是 1.25 位。然而，接收方对于 1001 代表 ABBA，ABC，DBA 还是 DC 却无法分辨。这条消

息的熵（根据上面的公式计算得出）是 $H = 1.75$，界定了编码是否能够解析的阈值。这条编码的平均长度是 1.25 位，低于此阈值，所以编码无法解析。

哈夫曼（Huffman）编码是一种快速在熵阈值内进行位编码的方法（图 3.5）。

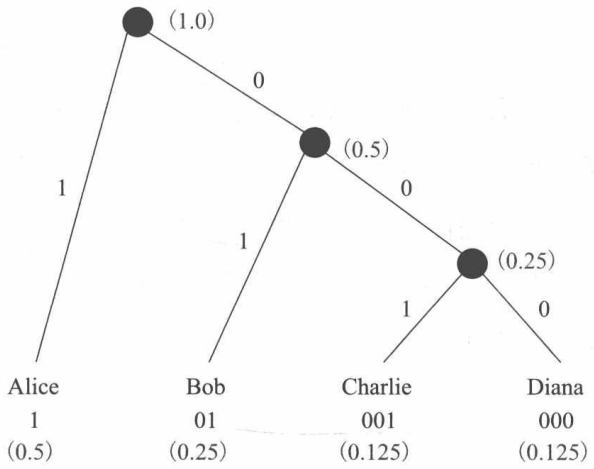


图 3.5　1951 年麻省理工学院的大卫·哈夫曼（David Huffman）设计了一套编码算法，在已知消息概率的情况下，此编码算法能够使得平均编码长度最小。该算法首先将每个消息都视为一棵单独的树。然后不断地将两棵具有最小概率的树进行合并，合并后树的概率为两棵子树的概率和。对于一个具有 n 条消息的编码，需要 n 次合并，形成最后的树。在本例中，Charlie 和 Diana 首先合并，然后他们合并后的树和 Bob 合并，最后和 Alice 合并。树中的每条路径定义了每一条消息的二进制位编码。哈夫曼的方法所生成的编码能够将平均编码长度控制在熵阈值范围内。若所有的消息具有相等的概率，则生成图 3.3 前面所示的编码，若概率不相等，则生成后面这种编码

从另外一个角度来看，熵的阈值定义了信道是否可信。如果消息源每隔 T 秒传送一则新消息，最短编码的平均长度是 H，那么这个消息源就产生了每秒传送 H / T 位的需求。如果信道的带宽是 H / T 位每秒或者更高，那么发送方所传送的所有位都能够到达接收端。若信道的带宽小于 H /T 位每秒，某些位就可能丢失，接收端就无法恢复原始消息。

46
~
47

文件压缩是信息论的一个重要应用，因为其可以减少存储空间和缩短传输时间。在大部分计算机程序中都使用标准码来表示文本，包括传统的固定长度编码 ASCII 和现代的变长编码 Unicode。这两种情况下每个字母都使用了相同长度的编码，因而，通过寻找重复模式并基于文件上下文用更短的编码代替这些模式，文本文件可被大幅压缩。例如，一个包含很多字母“e”的文件，可以用新的更短小的编码替换它的编码来压缩文件。新的编码取决于“e”在文件中的出现频率——在“e”频繁出现的文件中，这个编码可能

是3位，而在“e”不那么频繁出现的文件中，这个编码可能是5位。文件压缩算法会生成一个新编码到原始编码的转换表。“.zip”和“.rar”格式就采用了这种策略。这种压缩策略的设计也不会将信息“压缩”至低于熵的阈值。因为若是低于阈值，则无法保证完全
48 恢复原始信息，这种策略被称为*无损压缩*。

另一种策略是*有损压缩*。有损压缩方法具有更大的压缩率，但无法完全恢复原始文件。例如，MP3音频压缩通过丢弃绝大部分人听不到的频率来压缩音频，其压缩率大约为10，但是没法恢复丢弃的频率。JPEG图像压缩舍弃了部分人眼基本会忽略的颜色信息位，当然也没法恢复这些原始的图像位。这些压缩方案使得DVD、在线电影和唱片等能够以更低廉的价格卖给消费者。这些方法在感知质量上的小小损失，对于减少文件大小而言通常被认为是一种很好的折中。

信息的转换

一个单纯的通信系统只是简单地将信息从一处传输到另一处，但是计算机会做更多的工作，即转换信息。转换就带来了更多可能，其中最显著的产品就是新信息的出现。简单的转换包括将一个数平方、计算 π 至指定小数位数、对一列数字按照升序排列，每一次转换都是将一种信息模式作为输入，并创建一种信息模式作为输出。

因为二进制模式可以被解析为数，所以一次转换在数学上看来就像是一个输入数到输出数的映射函数。能够被机器计算的函数被称为可计算函数。图灵和他同时代的人们用这个概念来定义计算。图灵表示一个简单的抽象计算机——图灵机，足以实现任何可计算的函数（Turing 1937）。图灵机遵循一个非常简单的指令程序来实现这些转换。因为每一条指令显然可以被机器实现，那么计算机转换二进制的时候并不考虑它们的含义。这类似于香农所强调的，信道可以被设计得完全可靠并且不考虑传输信息的含义。

当我们稍微深入了解机器如何转换输入时，就会发现程序设计的一个重要方面，称之为*含义保留*（meaning-preserving）。设想两个数 a 和 b 相加，两个数相加意味着什么呢？这意味着我们要遵循一系列加法算法的步骤。这个步骤包括连续对 a 和 b 的两个对应
49 的数字位相加，然后将其进位结果传递到更高位。我们对于属于 $\{0, 1, 2, \cdots, 9\}$ 中的数字对有明确的加法规则，并且会产生进位0或者1。当为这个算法设计程序时，我们需要注意让每条指令都能够产生正确的加法结果。如果成功了，我们可以相信机器能够正确做出两个数的加法。若失败了，我们会说机器坏了。

换句话说，设计过程本身就是将我们脑中的加法过程转换为机器执行加法的指令模式，加法的含义就存在于机器和其算法的设计之中。

这对于任何其他的可计算函数也是适用的。我们把大脑中想法的具体含义转换成可计算的函数，将它输出到程序中，从而控制机器实现这个想法。这个过程也就是将具有我们想法含义的函数转变为对机器的设计。

从这个角度看，机器和通信系统不考虑二进制数据含义进行信息处理的概念有些站不住脚。算法和机器被工程师和程序员植入了含义，我们设计机器从而使得每个计算步骤和输出都是我们想要的含义，假设输入也具有我们想要的含义。仔细精确的设计就是为了不用担心机器曲解我们想要表达的含义。

计算机将计算函数和信道结合，一个信道将输入送到执行计算函数的机器，另一个信道将输出送到目的地。这种情况下信道和计算机似乎只完成了运输和操作信息位的工作。然而从人类的角度来看，计算带来了新的信息，无论计算机输出比输入更多还是更少的信息。例如，前文提到的一个计算 π 的函数，在输入一个三位数“900”时能生成 900 位的 π 的值，输出扩展了 300 倍的信息量。一个排序函数的输出和输入具有同样的位数，但是具有不同的顺序。一个平均函数能够计算出少量的数字来代表大量数据的平均值。

数字表示和机器操作都依赖于物理过程，每个机器操作都需要花费少量的时间和功耗。有些我们希望计算的函数需要花费太多的步骤以至于在有生之年无法等到机器返回结果。计算所需的物理处理严重限制了我们能够计算的内容。

计算的逻辑也会带来限制，其中最有名的就是我们想计算但无法计算的函数。图灵 50
在 1936 年提到了停机问题——不存在一个程序，可以检查任何程序并且判断其在给定输入上是否包含无限循环。一个现在的实例是恶意软件检测，即不存在一个程序能够检测任意给定程序是否嵌入了恶意软件检测程序。我们在第 6 章检验计算的物理和逻辑局限性的时候，还会再回到这个话题。

即使当我们将注意力限制在可以计算并且很快返回有用结果的函数，也可以发现一些有趣的问题。当一个函数在计算我们没有见过的数据位时，这些数据位是新的信息吗（见图 3.6）？或者它们只是现有信息的结果？ DNA 包含信息吗？很多生物学家认为包含。若 DNA 是一种信息，那它的信息源和接收端又是什么呢？如果对 DNA 解码，我们能得到什么信息呢？解码的 DNA 可以用来寻找治愈遗传疾病的治疗方案，也可以识别犯罪现

场的行为人。将 DNA 与数据库匹配仅仅揭示了现有的信息还是生成了新的信息呢？经典的信息论无法回答这样的问题。

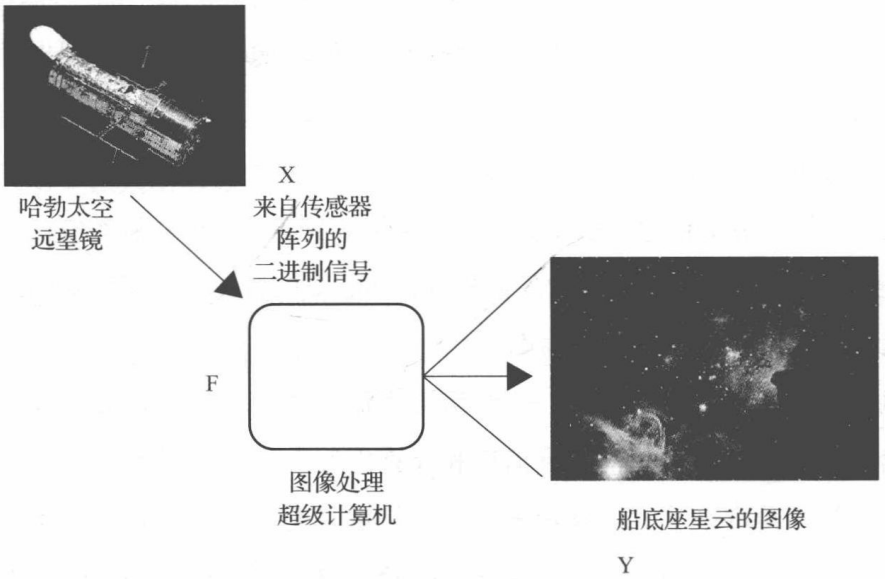


图 3.6 哈勃太空望远镜的集光传感器阵列编码 TB 级数据并传输到地球，这些数据再被处理成为图像。计算理论将这个图像处理过程描述为方程 $Y = F(X)$，其中输入数据 X 在函数计算后产生输出图像 Y。这个机器实现的功能（将信号发给它，它又生成信号）不依赖于望远镜传输的信息的含义，然而人类看到 Y 输出的是一张美丽的图片——船底座星云的实际展示（图片来源：NASA）

交互系统

许多计算机程序是交互系统：接收新的输入，在很多点上生成新的输出，除非被干预或者程序崩溃，它们可能会无休止地做这些工作。交互系统无处不在，每一个操作系统都是一个交互系统，比如一辆汽车的 GPS 系统、Facebook、一个在线商家的服务器，或者互联网的域名解析系统。互联网本身也是一个全球的数据交换和协调行动的交互系统。交互系统的一个显著特征就是持续不断地运行，同时没有程序结束。相反，函数系统在计算出答案后，程序结束运行。

多年来，计算机科学家们激烈地争论交互计算是否比函数计算更强大（Goldin et al. 2010）。近年来专家们一致同意交互式计算更加强大。作为当代的难题，如何有语义地为数字图像打标签解释了这个原因，因为这个问题的解决方法依赖于交互：游戏构造了人机交互来实现人和机器都无法单独完成的功能（见图 3.7）。交互系统所生成的输出是已知机器无法做到的。

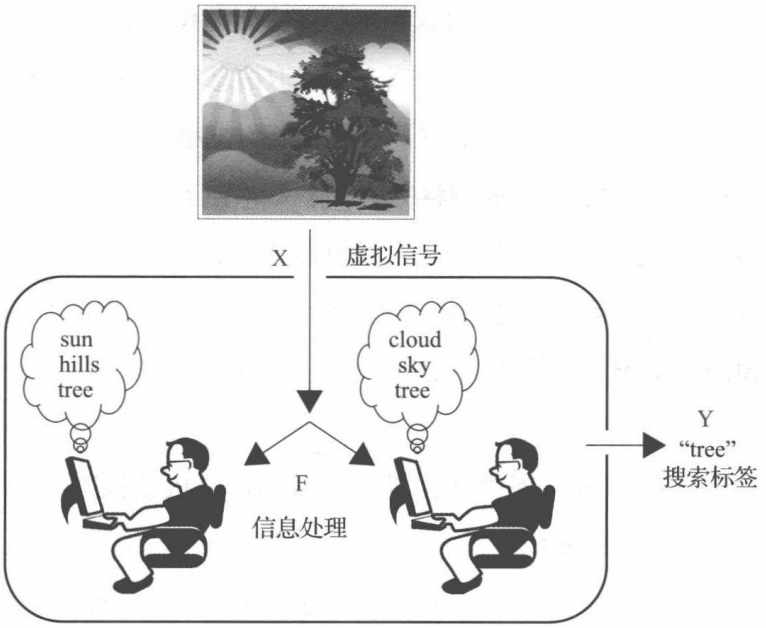


图 3.7　在 2004 年卡耐基梅隆大学（CMU）的 Luis von Ahn 和 Laura Dabbish 发表的一篇论文中，描述了一个新颖的电脑游戏 ESP。在游戏中，玩家被两两分组，给予同一张图片并要求用单词来描述这张图片的内容，游戏的目标是找到同伴（不知道也看不到）也使用过的相同单词。公共的单词就成为这张图片被搜索时的新标签。游戏将人和电脑组队去计算一个图片识别函数，没有人知道如何单独使用计算机进行计算。像其他函数一样，这个函数也转换信息，不过这个含义现在是由玩家来交互地完成

解决悖论

在上面的讨论中，我们注意到一些关于信息的看似矛盾的结论：

1）工程师设计的通信系统能够在不理解信息含义的情况下运行，那么人类信息接收者如何收到含义呢？

2）工程师所设计的计算机系统能够在不了解程序与二进制模式数据含义的情况下转换信息，那么这些系统如何生成新的信息呢？

3）程序员设计的程序能够在对其数据无知的情况下运行，那么人类用户是基于什么有意义地解释程序的输出呢？

4）建立连接（如放置一个网页链接 [Berners-Lee 2000]）会产生新的信息吗？

5）如果设计一个计算机程序来生成欺骗性信息，那它的输出还算是信息吗？

6）加密消息中的信息又在哪里？

7）如果一个计算机程序完成了新的科学发现，那么它是创造了新的信息还是仅仅传递了人类不了解的已知信息呢？

51
~
53

信息学常见的理论和观点都不能回答这些问题。比如，我们说符号“携带”信息，这只会引出新的问题，携带的信息在哪里？如何嵌入和提取信息？另一个例子是嵌入在社会传统中的含义通过机器的输出所触发，这种含义只会有助于引发第三个问题。还有另一个例子是“标记对象”模型（Rocchi 2012），该模型将标记和含义将结合，但仅对显式信息有效。

含义保留的变换解决了所有的悖论。人们有意设计出软件程序来支持实践、建立连接、隐藏信息或者寻找新的信息，这些含义是设计师设计机器反馈的所有结果，因此人类用户会认为这是预期的反应。当某个机器没有给出预期的行为时，设计师和用户都会认为它损坏了。

这些悖论总会在初期显现，因为计算机科学家总希望机器拟人化，比如，我们说一个机器能够理解它的输入，或者机器对于输出很有创造性。而当别人深入了解我们的程序和机器时，他们只能看到机械步骤。这些步骤中他们看不到“理解”甚至“创造性”。理解和含义来源于设计者，是他们设计了机器的模式并在使用时产生预期的含义。

信息和发现

当我们说计算机发现了新的模式时我们想说明什么呢？设想一个程序能够发现数据的趋势，首先提供一组在以往性能实验中观察到的输入输出对（x，y）给程序，然后利用统计回归，程序找到了最好的参数 a 和 b 表示一条直线来拟合这个数据：$y = ax + b$，程序的输出是直线的常规表达。这个输出对于了解如何利用直线进行预测的用户是有意义的。很容易设计另一个程序来使用具有参数 a 和 b 的直线来预测 y，而 y 是由新的输入 x 生成的。

这就是一个设计师使用数学知识从一系列数据中计算最佳拟合参数的过程，计算中的步骤是机械的，输出对于那些了解数据中直线趋势模型的人们是有意义的。这些意义来源于设计师，而不是数据的处理。

对于不了解数据中直线模型趋势的人，也就不知道输出的含义。但是这并不意味着
54
输出的含义是主观的，只意味着设计师没有打算让程序对这些用户产生任何意义。

在 20 世纪 80 年代，研究者开始使用强大的计算机筛选大数据集来试图发现一些模式。他们使用贝叶斯（Bayesian）推理（这是一个复杂的数据分析方法）来推算最有可能产生数据的一系列条件。贝叶斯推理基于贝叶斯条件概率统计公式：

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}$$

它说明的是给定证据 E，假设 H 的概率就是：给定假设 H 后该证据 E 的概率，乘以假设 H 的概率，再除以证据 E 的概率。图 3.8 给出了一个简单的示例，医生已知病人头疼，试图诊断他是否患有流感。 55

这种情况下的发现是一个新的假设。程序能够生成一系列假设，然后根据手中已知的情况，按照贝叶斯定律计算每一个假设的概率，把其中最有可能的假设作为这个“发现”。

在这种情况下，设计者结合贝叶斯定律和搜索方法，在给定数据下找到最大可能的假设。这个程序的输出就是那些明白假设和数据“含义”的用户所预期的，用户将决定是否将这些假设视作一个发现。

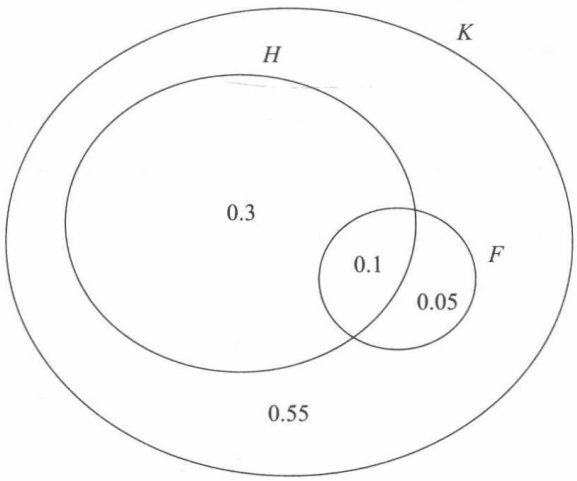


图 3.8　韦恩图演示了如何利用贝叶斯定律评估一个很难判断的假设。在所有人口的集合 K 中包含一个子集 F，代表患有流感的人，还有一个子集 H，代表患有头疼的人。医生看到一个病人抱怨头疼，担心自己患上流感。根据贝叶斯定律 $P(F|H) = P(H|F) \cdot P(F) / P(H)$。医学数据告诉医生，患有头疼的概率 $P(H) = 0.4$，患有流感的概率 $P(F) = 0.2$，在患有流感的人中患有头疼的概率是 $P(H|F) = 2 / 3$，因此 $P(F|H) = (2 / 3) \cdot 0.2 / 0.4 = 1 / 3$，即三分之一。在没有任何信息的时候，患有流感的概率是 0.2，但若已知患头疼，患有流感的概率则上升到 0.33

在经典的信息论中，我们说贝叶斯推理发挥作用，是通过已知消息源的数据来决定消息源的内容。在消息通信中，香农合理假设消息源的内容为先验信息。在科学发现中，消息源中包含的一系列概率最初是未知的，推理过程使得消息和它们的概率可知。贝叶斯推理是一个自动将消息源中的观察数据转换为消息源内容的方法。

总结

自古以来，人类就开始编码信号，使其可以在不同的媒介中进行传输。在 20 世纪 40

年代，香农的信息论提出了四大准则：

- 每一个通信系统都可以被建模成一个噪声通道，其携带着被编码的信号，该信号表达着消息源中的消息。
- 消息源的熵决定了消息源最短的可解析编码的长度，哈夫曼编码是和熵一致的一种编码（信息熵小于一个比特）。
- 充分冗余的比特位能够添加到任意编码中，以此来克服信道中的噪声，并且能同时保证百分之百准确的接收。
- 利用更短的编码替换模式，文件能够被压缩到更小的尺寸。

这些原则使得通信和计算机工程师能够设计数字系统，这些系统在传输过程中不会丢失信息，并且噪声引入的错误也可以被消除。

一些不理解香农理论的企业家认为“比特而非原子”的理论能够预示一个激增的新兴经济体，但他们的梦想是不会实现的，因为真正的通信和计算系统的基础来源于将数据
[56] 记录表示成物理信号和状态，而计算和传输需要消耗时间和能量。我们在计算上花费了很多能量：互联网的连接点和数据中心消耗了世界近 6% 的电力。我们不能寄希望于假定能通过比特而不是原子来解决棘手问题就忽略了这个问题：比特背后的原子也是很重要的。

正如摩尔定律所预示，我们存储信息的能力成指数增加，我们阐明信息含义的压力也在不断地增加，而香农信息论的定义无法解决这个问题。这看似是个悖论：系统如何在不考虑信息含义的情况下处理信息，同时还能在用户的体验中产生含义。

含义保留的变换融合了机器的无意义机制和能让机器产生含义的人类经验。程序设计师编制指令，从而能在用户群体产生预期含义的输出；每一条指令的执行，逐步将部分计算结果接近预期的输出。让我们为程序设计师这个角色鼓掌，是他们给了我们有意义的软件和硬件。

致谢

这一章改编自 Peter Denning 和 Tim Bell 的“信息悖论”（2012），其发表于《美国科
[57] 学家》。

第4章
Great Principles of Computing

机　器

当一个人拥有了纸、笔和橡皮，并且遵从某种规则，就成为一台广义上的计算机。

——Alan Turing

计算机可能是一种真正意义上的人性化驱动力。机器完成的工作使得某种生活成为可能，而人类做的工作则使得生活变得更有意义。

——Isaac Asimov

计算机科学家喜欢对事物进行抽象化。抽象是一种思维模型，它抓住事物的一些最本质特征，而忽略其他所有不太重要的特征。计算机科学家经常将编程描述为设计一个层次化抽象的过程，它由一组“抽象对象”和用来操作这些“抽象对象”的一组特定函数构成。这种说法非常流行，以至于计算机科学常常被人们吹捧为最擅长于进行抽象的学科。

和其他领域常见的各种数学抽象相比，计算抽象有一个重要的差异：计算抽象可以执行操作。与此同时，“抽象”这个术语也很容易让我们忽略一个事实：计算操作是要通过由程序控制的实际物理过程来实现的。

我们以音乐为例加以说明。在计算机中，一首“歌曲”是通过一个 MP3 格式的文件来表示的，它包含了出版商提供的这首音乐的数字化信息。当我们要听这首歌曲时，可以对这个文件执行一个叫做“播放”（play）的程序。“播放”程序读取存储于磁盘上的这个 MP3 文件，将其中的数百万个比特转化成电信号，然后传输到耳机。在那里，扬声器的电路会驱动隔膜产生振动，从而发出声音。从抽象的层面来看，播放程序和 MP3 文件都是单独的对象：对“歌曲”应用“播放”，你就能听到音乐。上述过程的具体实现是非常复杂的，它涉及大量步骤，每一个步骤都依赖于一个具体的物理过程。

本章讨论如何从物理上来组织机器，使得它可以计算函数（function）。机器所能执行的每步操作都被表示为逐个单独的指令（instruction），例如对两个数求和。一个程序（program）就是按某种方式精确排列好的一串指令，它使得机器可以得出我们想要计算的函数的数值。指令和数据都以二进制的形式被编码保存于存储器中。当指令序列被读入到处理器（processor）中，它会驱使硬件进行操作，将输入数据转换成最后的输出数据。

在电子计算机发展的最初阶段，程序员都是直接用二进制形式来编写代码，并按顺

序排列到纸带上或者卡片上。随后，编程语言很快取代了二进制编码，因为使用编程语言可以大幅降低出错的可能性。通过采用一个叫做编译器的特殊程序，人们把通过编程语言编写的程序自动翻译成二进制的机器代码。在下一章介绍编程的时候，我们会讨论一个编译器是如何完成这些工作的。

计算机中各个部件的组织方式，通常被称为计算机体系结构[1]。一台计算机的体系结构规格涵盖了用于执行指令的中央处理器（CPU），用于存放程序代码和数据的随机访问存储器（RAM）[2]，以及在存储器中用来组织程序各个部分的数据结构。

机器

机器是一种可以利用能量来执行特定任务的装置。机器通常以机械的、化学的、热力的或者电子的方式来驱动。电子机器是一种通过电力驱动并且不包含运动部件的机器。这样的机器包括收音机、电视、手机以及平板电脑等。

自动机是一种能自我进行操作的机器。时钟上的布谷鸟，也曾经一度被视为一种自动机。同样，18 世纪末期的 Turk 国际象棋机也可算是一种自动机（图 4.1）。从 20 世纪 40 年代起，计算机科学家开始将自动机作为一种计算机的抽象数学模型。到 20 世纪 50 年代，科学家相信嵌入在软件或者机器人中的自动机拥有自我思考的潜力。

用于辅助计算的机器可以追溯到几千年前。早在公元前 2700 多年前，美索不达米亚、古埃及、波斯、罗马和古中国的商人就开始使用算盘来进行求和。古希腊人记载了如何通过测量影子来测量树的高度，即通过计算一根已知长度的木棍的影子和树的影子两者之间的长度比例，可以计算出树的高度。从某种意义上看，这根木棍以及这个计算过程就是一个简单的计算设备。而另一种测量工具——滑动尺，则在 John Napier 提出对数概念
60 之后的 1620 年前后被发明出来。常被大家戏称为“滑动的棍子”的滑动尺，是一种标准的计算机器。它一直被工程师广泛使用，直到 20 世纪 70 年代电子计算机出现之后才被取代。1642 年，Blaise Pascal 发明了一台可以进行数字加减法的机器，并提出了通过反复进行加减运算来实现乘法和除法的算法。Charles Babbage 设计了一种差分机器（1822—1842），用来计算常用数学函数的数值表。那时，已有的数值表都是通过繁冗的手工计算得到的，数据中常常含有错误，这给当时的航海家和其他使用者带来了极大的风险。1911 年，Marchant 公司开始销售由齿轮、滑轮和杠杆构成的，能完成加减乘除运算的机械计算器。1922 年，德国工程师 Arthur Scherbius 发明了一台名为 Engima 的用来生成密码的机器。波兰人在 1932 年破解了密码，并将这些信息传递给了英国人，英国人利用这些

信息在20世纪40年代初制造了名为Bletchley的密码破译机。在20世纪20年代后期，Vannevar Bush发明了差分分析器，通过机械的求积法来求解微分方程。

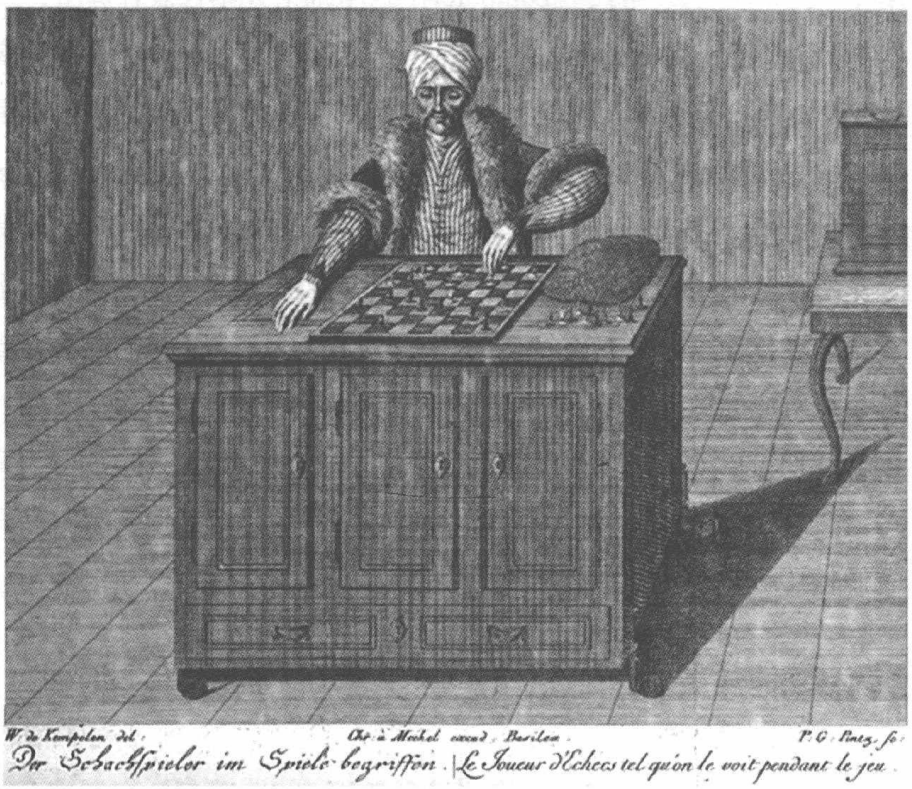


图4.1　Karl Gottlieb van Windisch在他1784年出版的书《Inanimate Reason》中描述了Turk，一台可以下国际象棋的机器。在1770年之后的长达84年里，这台机器的历任形形色色的主人都对外宣称这是一台可以和任何人下象棋的自动机器，而事实上它也确实赢得了大多数的比赛。然而这却是一场精心制造的骗局。事情的真相是，一名国际象棋高手就躲在这台机器的柜子里，通过镜子来观察棋盘上的棋子，然后通过杠杆来操作自己的棋子。这台机器给人的错觉，恰好迎合了人们内心深处的某种潜意识，抑或是某种恐惧，即人类的大脑只是一台机器而大多数的智力行为其实都只是机械的运动而已。在1997年，IBM制造的国际象棋专用计算机巨蓝（Big Blue）打败了国际象棋大师Garry Kasparov。人们对此的反应并不是认为机器拥有了智能，而是计算机比Kasparov搜索得更快而已

在第二次世界大战期间，美国军方在阿伯丁试验场委任了由女性组成的一些小组来计算火炮的弹道数值表。炮手们利用这张表，根据风速和炮击目标的范围，来确定最佳的射击方向和角度。根据写在纸上的程序，计算小组的女队员操作机械计算器（比如之前提到的Marchant计算器）来算出这些弹道数值表。手工计算很容易出错，而且随着新的军械不断被设计出来，人工计算再也无法满足大量的弹道数值表计算，因此美国军方决定采用电子设备来替代手工计算。1943年，在宾夕法尼亚大学，美国军方提出了建造第一台计算

机（即 ENIAC）的计划。ENIAC 在计算弹道数值时，运算速度可以比手工计算快一千倍。虽然这台机器直到战后的 1946 年才建成，但是从那以后，军方就开始大量地使用计算机。

有趣的是，在 20 世纪 20 年代“ computer”这个词语并不代表计算机，而是指计算员，一种从事数值计算行业的人。因此，当第一台电子计算机出现时，它被命名为“自动计算机”，以便和计算员区分开来。20 世纪 40 年代的第一台电子计算机，首字母缩写以“-AC”结尾，表达的就是这个含义。

1937 年，图灵将计算机定义为一种可以进行数学计算的机器。并且，他发现了一种无法被任何计算机计算出结果的函数。他使用“可计算性”这个术语来刻画一个函数是否能够通过计算机来进行计算。对于一个函数，如果存在一个有限的指令集合，当输入任何数值时都能够生成对应的输出值（见图 4.2），这个函数就称为是可计算的。例如，加法是可计算的，因为我们可以给出一个有限的指令序列，对于任何给定的 x 和 y，算出这两个数之和 $x + y$。在图灵时代，一个没有得到回答的数学问题是，如何表示所有可计算的函数的集合。我们将在第 6 章中更加深入地讨论这个问题。

61
~
62

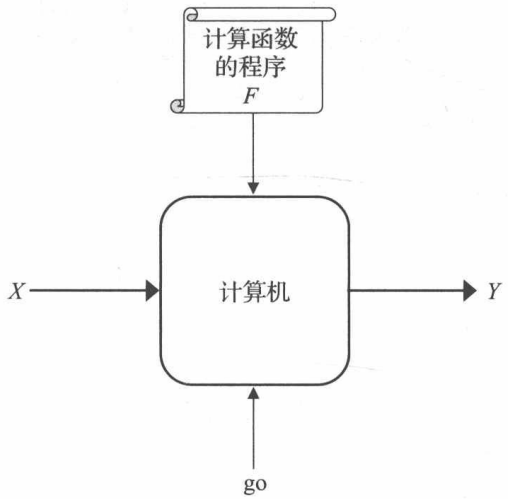


图 4.2 计算机是这样一种机器，它读取以二进制表示的 X 作为输入，经过计算之后输出以二进制表示的结果 Y。计算机是由指令来控制的，而这些指令被专门设计用于计算特定的函数 F。当一个信号到达图中的输入“ go”时，计算机开始工作。经过一段时间运行之后，计算机输出结果 $Y = F(X)$ 并停止。计算机完成计算并停止所需要的运行时间的长短，取决于所计算的函数和程序实现。有的程序可能包含无限循环，这会导致计算机无休止地工作下去。我们可以定义一个函数 $H(F, X)$，在给定输入为 X 时，程序 F 如果能够完成运算并停止就输出 1；反之，则输出 0。图灵证明了这个函数 H 是无法被任何计算机实现的

图灵认为，用来计算任何可计算问题的每一种计算算法，都是建立在一些非常基本的操作之上的，例如读取符号、根据读取的内容修改控制状态、记录符号。他创建了一种

抽象的机器模型，后来被称为图灵机。图灵机包含一个可以在一条无限长的纸带上游走的控制单元，它可以在磁带上的格子内写入和读出符号。这个控制单元的更具体行为，实际上是由机器的程序来规定的。机器的程序可以根据需要，通过使用循环来实现一些步骤的反复执行。同时，图灵还描述了一种广义的机器，它能够模拟任何其他的图灵机在给定程序下的行为。最后，图灵还证明，具有完备定义但不可计算的函数是确实存在的。比如，确定一个图灵机是否能够停机（而不进入死循环），就是一个不可计算的问题。和图灵处在 63
同一时期的几位科学家也设计出了几种其他的计算机模型，并证明了它们和图灵机是等价的。但是，只有图灵的设计最后成为大家公认的计算机参考模型，因为图灵机最大程度地模拟了真实电子计算机的工作方式，特别是处理器（控制单元）和存储器（纸带）。

计算机通常被定义成这样一种机器，它可以将输入转化为输出，然后停止。这种定义其实并不能反映计算机的全部用途。实际上，计算机和外界的交互是很普遍的。一台具有交互功能的机器，不断地接收到一些输入，并生成对应的结果进行输出，永不停止。我们在第 3 章中提到，一台与人类交互协作的机器，能够计算一些非交互式计算机所无法计算的函数。

可以计算的机器

计算机是由程序控制的一种机器，它根据所给的一个输入，计算出一个输出。接下来我们进一步来看看，如何能够建造一台按这种方式来工作的机器。

存储程序的计算机（stored-program computer），是一种实现了一组指令的电子硬件设备。*指令*（instruction）是指这种电子设备能够执行的一个单独的算术操作或者逻辑操作。*操作*（operation）是指最简单最基本的一种函数。最典型的操作一般读取两个输入，并产生一个输出。例如，加法（ADD）操作是对两个数求和，相等判定（EQ）操作是比较两个数是否相等，因此 ADD(3, 5) = 8，EQ(3, 5) = 0(false)。指令集合中通常还包含分支指令（branch instruction），这种指令用来控制当前指令结束之后，计算机跳转到何处去运行下一条指令。

程序（program）就是一串指令，它是根据需要计算的函数按照特定的设定来排列的。而*编程*（programming）则是一门设计程序并验证程序能够正确运行的艺术。

计算系统（computing system）是由计算机和运行于计算机之上的程序组合而成。计算机程序指挥计算机硬件来进行函数的运算。我们也可以说，计算机系统进行了函数的运算。

为了实现这一切，计算机系统需要以下部件：

1）一组精确定义的指令集，并通过硬件来实现这些指令。

2）通过指令序列来表示程序的具有精确定义的规范。

3）用于存放程序和操作数据的存储器。

64 4）能够按照程序预设的顺序读取和执行程序指令的控制器。

中央处理器（Central Processing Unit，CPU）是读取并执行程序中的指令的一种硬件设备，它按照程序中预先设定的顺序，每次读取和执行一条单个的指令。

随机访问存储器（Random Access Memory，RAM）是一种存储数据的硬件设备，数据存放在CPU能够读写的位置。RAM按照线性方式来组织管理存放位置。每一个存储位置都可以存放一个基本单元的数据，通常是一个字节（8比特）或者一个字（32比特）。这些存储位置按照0，1，…，直到2^n-1的顺序进行编号，其中n是这个存储系统中存储地址在二进制下的表示位数。RAM被称为是“随机访问”的，是因为它可以用相同的时间访问RAM中任意的一个存储位置。每一个存储单元都只存储二进制形式的数据（8位或32位）。RAM并不会试图去解读这些二进制的数据，它只会精确地存储和读取这些数值。存储器对CPU所发出的读写请求作出响应所需要花费的时间，称为*存储周期*。目前，典型的存储周期仅仅只有几纳秒。图4.3是CPU和RAM的示意框图。图4.4给出了CPU和RAM之间的一种接口示意图。

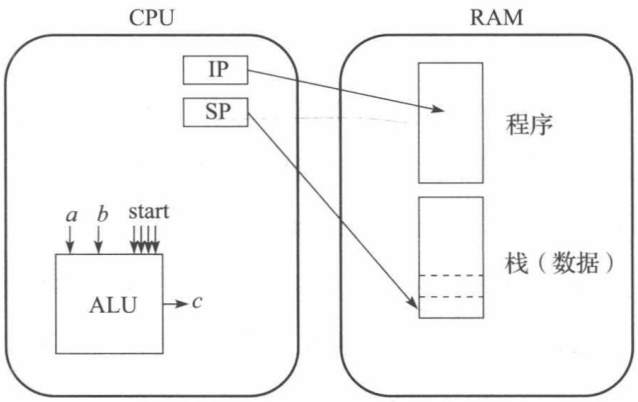


图4.3 计算机系统的硬件由中央处理器（CPU）和随机访问存储器（RAM）组成。程序及其数据都存储在RAM中，数据以栈的形式存放，也就是说新的数据被添加到栈顶，读取数据也只能从栈顶位置读取。CPU中有两个特殊的寄存器。指令指针（IP）寄存器用来存放程序中下一条将被执行的指令位于RAM中的存放地址。栈指针（SP）寄存器则存放栈顶元素在RAM中的存放地址。CPU还包含一个算术逻辑单元（Arithmetic Logic Unit，ALU），ALU接收两个输入数据（a和b）并产生一个输出（c）。ALU上有一组“开始”（start）信号线，通过这些线可以给ALU发信号来通知它执行哪个操作，例如加法、乘法、或者相等判断

65

实际的计算机系统中存储器不止RAM，例如还有硬盘。硬盘的访存时间并不是固定的，它取决于磁盘的寻道时间和转动磁存储媒介带来的延迟时间。关于在多种存储器之间移动数据的更多问题，我们将会在第7章存储部分进行讨论。

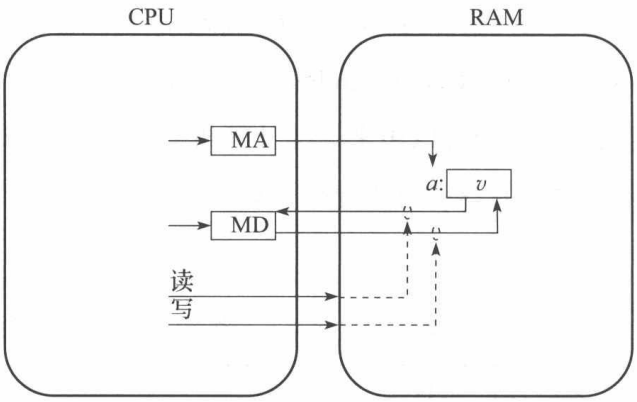


图 4.4　CPU 与 RAM 之间的接口由几个部分组成。接口设计的目的是为了使得 CPU 可以对 RAM 中某个特定的位置（a）进行读或写。读操作将某个选定的存储位置中的数值 v 传送给 CPU，而写操作则是将一个新的数值从 CPU 传送到某个选定的存储位置。内存地址（MA）寄存器负责通知存储器当前读写操作是要对哪个位置进行。内存数据（MD）寄存器负责存放读写的数据。读信号线通知内存硬件选中 MA 寄存器中指定的地址位置，并把这个位置中存放的数值传输给 CPU（存放到 MD 中）。写信号线则通知内存硬件从 CPU 读取数值（存放在 MD 中）并传送到 MA 寄存器中指定的地址位置。完成这一系列操作所需要的时间叫做存储周期。在现代 RAM 中，存储周期通常低于 10 纳秒

CPU 使用指令指针（IP）寄存器来记录下一条将被执行的指令的地址。它通过重复以下循环来执行程序中的指令，直到遇到程序中的退出指令时才结束这个循环：

1）**指令读取**，根据指令指针（IP）寄存器中的地址读取指令，并设置 IP = IP + 1。

2）**译码**，取出包含在指令中的操作码。

3）**执行**，执行指令对应的操作。

4）**检查**，检查中断：之前几步可能会出现错误等待处理。

66

CPU 中包含一个时钟，时钟每经过一个节拍就会发出一个信号。这个时钟信号会在 CPU 中传递并激活选中的电路。时钟节拍的典型间隔是 0.5 纳秒左右。CPU 需要四个时钟节拍来完成一条指令运行所需要的四个步骤。时钟节拍间隔长度的选择，取决于在指令循环的每一个步骤中所有电路进入一个新的稳定状态所需要的时间。如果时钟节拍间隔太短，一些电路来不及进入稳定状态，CPU 就会发生故障。图 4.5 展示了 CPU 如何进行译码和指令执行。图 4.6 展示了在 CPU 中算术逻辑单元（ALU）的 ADD 部件如何工作。

经过上述介绍，我们的结论是，可以设计一套电路使得机器执行一串指令从而计算一个函数。我们在这里所描述的设计，正是宾夕法尼亚大学、麻省理工学院、普林斯顿大学和剑桥大学在 20 世纪 40 年代为了联合建造第一台电子计算机而提出的设计方案。冯·诺依曼（一位和工程师一起工作的数学家）写出了这个设计方案的详细描述。正是因为他的这

一份里程碑式的文献，这个设计被称为“冯·诺依曼架构”，尽管这个设计实际上是由工程师 J. Persper Eckert、John Mauchly、Herman Goldstein、Arther Burks 和其他一些人一起发明的。

当然，也存在其他许多架构可以用来设计计算机。这些架构的共性在于，它们都把按照程序的指令来进行函数计算这一过程变得自动化。

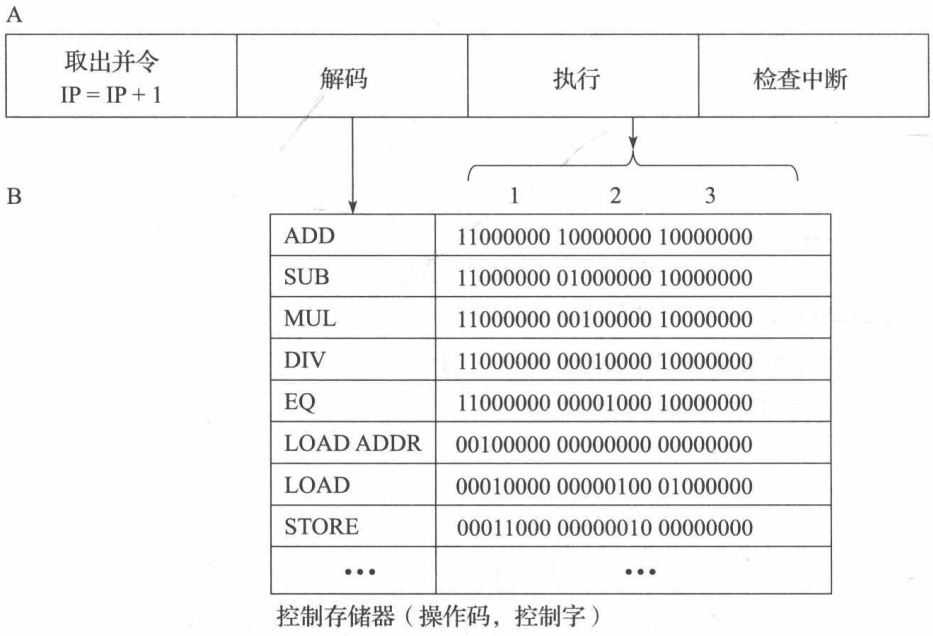
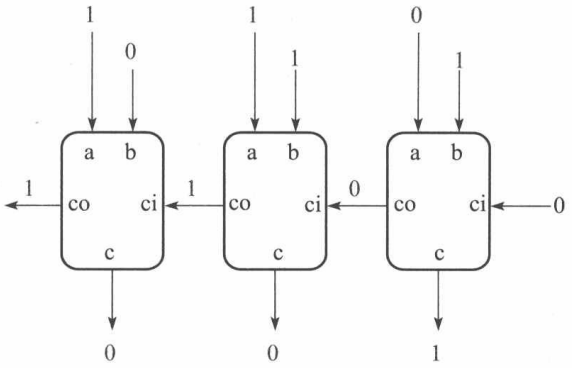


图 4.5 CPU 的指令周期由四个阶段构成（A）。在每个时钟节拍，CPU 进入下一个阶段。在第一个阶段中，CPU 从指令指针（IP，Instruction Pointer）指向的 RAM 地址中取出当前指令的一份拷贝，并把指令指针指向下一条待执行的指令。第二个阶段从当前指令中取出操作码，然后访问本地控制内存来获得当前指令的控制字（control word）。在本图的例子中，控制字包含三个八比特的块，分别对应于两个时钟节拍之间的三个时钟子节拍（subtick）。我们将控制字中的比特根据它们所属的块和位置进行标记；例如，比特 1.1 代表第 1 个块的第 1 个比特。在时钟子节拍出现时，控制字对应块中八个比特的每一位都会激活一个逻辑电路，因此最多可以有八个动作并行发生。本图的前五个例子中，指令都假设两个操作数分别在 R1 寄存器和 R2 寄存器中。比特 1.1 控制将 R1 的值复制到 ALU 的“a”输入中，比特 1.2 控制将 R2 的值复制到 ALU 的“b”输入中。控制字中第二块的前五个比特发送一个对应的触发信号给 ALU，告诉 ALU 执行加法（比特 2.1）、减法（比特 2.2）、乘法（比特 2.3）、除法（比特 2.4），或者是相等测试（比特 2.5）。控制字中第三块的第一个比特控制将 ALU 的输出复制到 R1 寄存器。图中的其他三条指令则激活不同的执行路径。LOAD ADDR 指令激活比特 1.3，代表的含义是“将指令字中的地址复制到 R1 寄存器”。LOAD 指令激活比特 1.4、比特 2.6、比特 3.2，分别表示“将 R1 寄存器的内容复制到 MA（内存地址）寄存器”，“激活内存读信号”，“将 MD（内存数据）寄存器的内容复制到 R1 寄存器”。最后，STORE 指令假设 R1 寄存器中存放了一个内存地址，R2 中存放了一个数值，它将会同时激活比特 1.4 和比特 1.5，然后再激活比特 2.7，它们分别表示“把 R1 寄存器中的值复制到 MA 寄存器”，“把 R2 寄存器中的值复制到 MD 寄存器”，“激活内存写信号”



ci, a, b	c, co
0 0 0	0 0
0 0 1	1 0
0 1 0	1 0
0 1 1	0 1
1 0 0	1 0
1 0 1	0 1
1 1 0	0 1
1 1 1	1 1

图 4.6　当对数字做加法时，我们采用一个很简单的算法：先对个位的数字求和，然后对十位的数字求和，再对百位的数字求和，以此类推。同时，当某一位求和的结果大于 9 时，则向相邻的更高位进行进位。比如，为了计算 17 和 26 的和，我们从最低的个位开始，7 加 6，得到的求和结果为 3，同时向更高位进位 1。然后我们计算十位上的和，1 加 2 同时加上进位 1，得到求和结果为 4，并且没有进位。因此两数的和为 43。对于二进制的数值系统而言，上面的算法变得更加简单，因为二进制数字的和只能是 0 和 1。图中描述了一个输入为三个比特的二进制加法器（左侧），输入为数字 a 和 b，输出为数字 c。进位信号（ci = 从低位输入的进位，co = 向高位输出的进位）从低位向高位依次传递。从右向左看，第一个加法器计算个位（1 的倍数位），下一个加法器计算第二位（2 的倍数位），第三个加法器计算第三位（4 的倍数位）。某些情况下的比特组合会产生进位信号，比如 1 + 1 = 0 会产生进位 1。上图右侧表示了所有可能的二进制输入组合，以及对应的二进制输出组合。我们将最左侧的进位信号（co）作为输出的第四位，因为有可能求和结果会大于 7（三位比特所能表示的最大数）。最大的和可能是 111(=7) + 111(=7) 等于 1110(=14)。图中的每一步计算都是由晶体管来实现的。当完成每一位运算的晶体管都进入稳定状态时，我们就得到求和的结果；而进入稳定状态所需时间的最坏情况是，进位信号从最低位一直沿着图中的进位链条传递到最高位。对大数进行求和也采用这个相同的结构，32 位的机器用 32 个比特来表示一个数，并采用包含 32 步的加法器。在大多数的计算机中，加法器都是构成更大的算术逻辑单元（ALU）的基本部件，可以完成加、减、乘、除，以及逻辑测试操作，比如大于、小于和相等判断

程序及其表示

前面的这些讨论可能给我们一个这样的印象：一个程序可以是由机器指令集中的指令按任意方式组合而成的一个序列。但事实不是这样的，程序必须遵守一些结构化的规则。在程序中，每条指令具体完成的操作，以及每段代码整体上完成的功能，都不能具有歧义，否则我们无法确保计算机对于同一个输入在每次运行时都可靠地得到同一个输出。

让我们来概述一下程序的设计。通常来说，我们在做计算的时候会做以下三类事情：

1）按照一个严格的顺序串行地执行指令（顺序）。

2）根据一个测试操作的结果（真或假），在两个计算选项中选择一个执行（分支）。

3）对某个运算重复执行多次，直到某个终止条件成立时停止重复（循环）。

67～69 注意上面提到的迭代模式有可能导致一个无限的循环，因为终止条件有可能一直无法满足。

编程语言（programming language）是一个语法规则的集合，它精确地描述了前面提到的各种程序结构的表示方法。目前已经有上千种编程语言，虽然编程语言是多种多样的，但它们都有一个相同的目标：即描述如何让一台计算机来计算某个特定的函数。

当设计程序时，我们可以设想有一个指针，它沿着程序中的指令逐条地移动，每移动一次，机器都执行指针当前所指向的一条指令。这个指针叫做指令指针（IP）。CPU 将指令指针实现成一个寄存器，它里面存放的是程序中将被执行的下一条指令在 RAM 中的地址（见图 4.3）。当我们执行完一条指令之后，通常会按着顺序继续执行在指令序列中的下一条指令（IP + 1），除非遇到了一条控制指令将 IP 进行了重新设定。比如，一条“GO 17”指令会将 IP 的值指向 17，因此 CPU 接下来将会执行内存中位置为 17 的那条指令。

关于机器如何执行程序，我们接下来将讨论如何设计指令集，使得机器可以支持任何遵守上面所提到的三类结构的程序。

栈式计算机：计算机系统的一种简单模型

几千年来，学生在学习代数的时候一直都不断被提醒：算术运算是有优先级次序的，要先算乘除法，然后再做加减法。对于具有同等优先级的运算，则应该从左到右依次计算。这些规则保证了，无论是谁来做这样的运算都能得到同样的结果。例如，根据优先级次序，表达式 1 + 2 × 6 / 4 − 2 会被依次计算为：

1 + 2 × 6 / 4 − 2

1 + 12 / 4 − 2

1 + 3 − 2

4 − 2

2

更高年级的学生可能还学过第三种优先级：指数和对数运算。这两种运算的优先级比乘除法的优先级更高。

同时，在学习代数的过程中我们也学过，可以通过加括号的方式来强制某一步运算优先进行，而不受相邻运算符优先级的干扰。例如，将上述表达式的最后两项放进括号，就能得到一个完全不同的结果：

70

1 + 2 × 6 / (4 − 2)

1 + 2 × 6 / 2

1 + 12 / 2

1 + 6

7

1924 年，波兰的逻辑学家 Jan Lukasiewicz 发明了一种新的表示方式，现在称为逆波兰表达式（RPN）。它遵从运算符的优先级，同时又避免使用括号。这种表示方法的基本思路是在两个操作数后面放置一个操作符，从而构成一个表达式。按照逆波兰表达式将上面两个式子表示如下：

1 2 6 × 4 / + 2 −

1 2 6 × 4 2 − / +

在计算机科学发展的早期，有人注意到，波兰表达式可以通过栈的形式来计算。栈是一种后入先出的内存结构。我们按照从左到右的方式来读取波兰表达式，当遇到数字的时候，就将数字压入栈顶，遇到运算符的时候，则将栈顶的两个元素取出，根据运算符进行运算，并把运算结果放回栈中。例如，对于上面的第一个表达式，在计算过程中对应的栈状态，可以像下面这样表示（栈顶在右侧）：

1　　（将 1 压入栈）

1 2　　（将 2 压入栈）

1 2 6　　（将 6 压入栈）

1 12　　（2 和 6 出栈，做乘法，并将结果 12 压入栈）

1 12 4　　（将 4 压入栈）

1 3　　（12 和 4 出栈，做除法，并将结果 3 压入栈）

4　　（1 和 3 出栈，做加法，并将结果 4 压入栈）

4 2　　（将 2 压入栈）

2　　（4 和 2 出栈，做除法，并将结果 2 压入栈）

Burroughs B5000 计算机（1961）以栈的形式组织内存，实现了对表达式进行高效计算的方法（Organick 1973）。英国的电子 KDF9（1963）也使用了栈式的结构。Hewlett Packard 科学计算器 HP-67（1972）也同样使用了这种结构，因为这样可以减少在计算复杂表达式时键盘的敲击次数和错误。现代的 HP 计算器继续沿用这种栈式的结构。从 Algol（1958）开始，许多编程语言都是基于一种假定而设计的，即它所使用的机器具有栈式的 71

内存结构。现代的多核计算机芯片也将栈式内存结构用于子过程的调用。现代的编译器则使用 CPU 的寄存器来模拟压栈操作，从而计算表达式。栈式存储结构几乎无处不在。

表 4.1 展示的是为图 4.3 所描述的 CPU-RAM 架构而设计的一个指令集。“操作码”列中是每一条指令的缩写。指令的执行效果在“执行前”和“执行后”两列中，显示的是在指令执行前后栈的结构变化。字母“ S”代表了当前指令执行之前栈的状态。Mem[a] 表示存储在内存位置 a 中的内容。在“内存效果”列中给出了某个内存位置中值的改变以及指令指针的值的改变产生的效果。

表 4.1 栈式机器的指令集

类型	操作码	名称	执行前	执行后	内存效果
算术和逻辑操作	ADD	加	Sab	Sc	
	SUB	减			
	MUL	乘			
	DIV	除			
	EQ	相等测试			
	NE	不相等测试			
内存接口	LA a	加载地址	S	Sa	V = Mem[a]
	L	加载	Sa	Sv	使得 Mem[a] = v
	ST	存储	Sav	S	
排序	GO	跳转	Sa	S	使得 IP = a
	GOF	为假时跳转	Sav	S	当 v = 0 时使得 IP = a
完成	EXIT	退出	empty	empty	

图 4.7 展示了用表 4.1 中指令集编写的一个程序例子，这个程序实现了将一个表达式的值赋予变量 X 的过程。

X = A*(B + C)

	LA X	LA A	L	LA B	L	LA C	L	ADD	MUL	ST
	X	X	X	X	X	X	X	X	X	
		A	4	4	4	4	4	4	20	
				B	3	3	3	5		
						C	2			

图 4.7 这一系列的快照展示了在实现赋值语句 X = A*(B + C) 的过程中栈的使用情况。在这个例子中，A = 1，B = 3，C = 2。当程序执行完毕之后，内存中 X = 20，栈变为空

过程与异常

在机器的指令集中包含了一些在运行程序时控制 CPU 执行顺序的指令。更高级的一

些编程语言往往要求更复杂的流程控制。因为除了直接使用指令集，它们还允许程序员编写一些自己定义的函数，以及编写函数来处理错误或者其他需要特别关注的事件。支持程 72
序员实现这些功能所需要的基本结构，就是过程的调用和返回机制。过程调用机制的目的在于，将 CPU 转到另一个程序的首条指令处开始执行，而当该程序执行完毕之后，再将 CPU 返回到刚才启动调用的代码位置。

第一台存储程序式计算机的设计者意识到，程序员希望在程序中添加自己编写的函数，将这些函数实现成一些子程序，并希望它们可以像最基本的机器指令那样很容易地被调用。子程序机制允许程序员在任何需要的位置去调用一个事先写好的子程序，而不是将子程序的代码在调用处重写一遍。同时，它也使得专业人士可以为一些标准的函数建立专门的程序库，比如三角函数和代数函数，这些程序可以被任何其他人可靠地调用。

在 20 世纪 50 年代，可重用的代码最开始被称为*子程序*。20 世纪 60 年代，在 Algol 语言的影响下，这个名字变成了过程（procedure）。一个过程通常就是实现了简单而单一的某个功能的子程序。

过程的核心思想在于，它仅仅在调用发生之后并且返回之前，才是处于活跃状态的。并且，处于活跃状态时，它所需要的数据都存储在一个私有的内存区域中，这个内存区域称为*活动记录*（Activation Record，AR）。当一个过程被调用时，计算机会先为这个过程分配一块内存区域，用来存放这次调用的活动记录；而当过程返回时，则会释放这些内存。当过程的调用发生嵌套时——即一个处于活跃状态的过程又调用了其他的过程（包括它自己）——就会出现多个活动记录，每个活动记录对应一次调用。这些活动记录会按照调用发生的顺序串联起来。因此，当一个过程返回时，程序就能回到调用发生的位置，继续往下执行（见图 4.8）。由于返回发生的次序正好和调用发生的次序相反，活动记录会在调用发生时压入到一个常规的栈中，并在调用返回时出栈（见图 4.9）。 73

现在来看关于对 LOG(Y) 函数进行调用的一个实例。这个过程调用的意图是执行一段计算 Log_2Y 的代码，并在栈顶返回计算结果。为了实现这个意图，CALL 指令将 CPU 交给 LOG 函数的代码。LOG 函数的代码计算出结果，并将结果放置于 LOG 函数活动记录中预留的起始位置。当 LOG 函数的代码结束运行，它会执行一个 RET（返回）指令，这使得 CPU 回到 CALL 指令的下一条指令继续向下执行。下面五个步骤更详细地阐述了这个过程是如何实现的：

1）调用者根据 LOG 函数的活动记录模板，在栈顶建立一个新的活动记录。这个活动记录为参数（Y）预留了一格存储空间，并为两个局部变量预留了两格存储空间。最后

通过一系列（如 $k = 6$ 次）装载操作来填充这些存储位置，以完成活动记录的创建。

2）调用者将 LOG 函数的地址置于栈顶。这时，新的活动记录的起始位置处于栈指针向下 k 格的位置，也就是说新的活动记录的起始位置是 SP − k。

3）调用者执行指令 CALL k，即执行以下步骤：保存 IP 和 AR 寄存器的内容到为它们预留的位置（分别在位置 SP − k + 1 和 SP − k + 2）中，设置寄存器 AR = SP − k，并将栈顶的数值出栈并存入寄存器 IP。（现在 IP 寄存器已经指向 LOG 函数入口。）

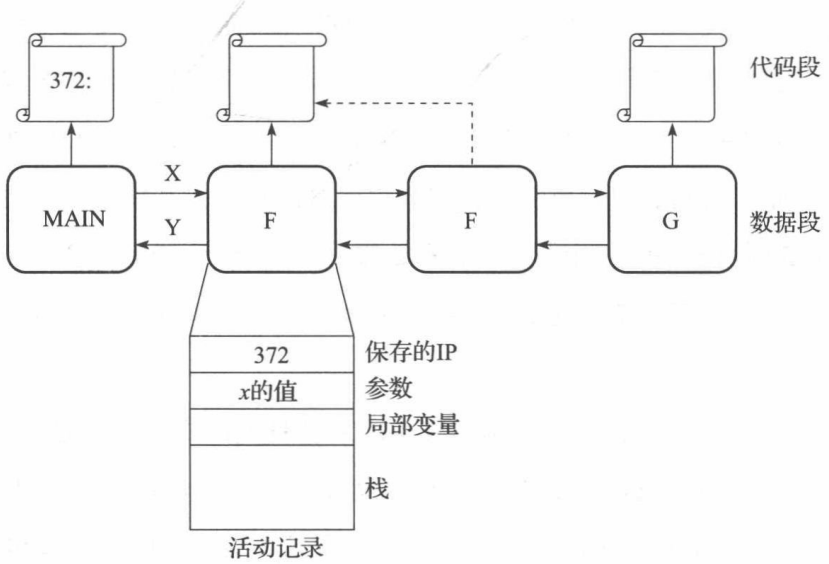


图 4.8 编程语言提供了过程（独立的子程序）来实现函数。MAIN 函数可以看做是被操作系统调用的一个过程。在这个例子中，MAIN 函数调用了过程 F，之后过程 F 调用了它自己，然后第二个过程 F 调用了过程 G。当过程 G 被激活并运行时，过程 MAIN 以及对 F 的两次调用都处于活跃状态，但是被暂停和悬挂起来了。图中向右的箭头表示调用动作，以及参数的传递；例如 MAIN 函数调用 F 时传递了参数 X。向左的箭头表示了返回的数值，比如 Y = F(X) 的值。每个过程都是由一个代码段和一个数据段实现的。虚线箭头表示了每次调用对应的过程代码。当过程 F 调用它自己的时候，每个 F 调用实例都有一个属于自己的数据段，但是所有对 F 的调用实例都共享过程 F 的同一段代码。数据段被实现成一项活动记录，它包含了下面几部分：保存的指令指针（IP）、过程调用的参数、过程使用的局部变量，以及一个仅供该过程使用的栈内存区域。保存的 IP 指针属于调用者；例如，MAIN 函数在代码段中地址 372 的位置调用了过程 F，当 F 执行完毕之后，CPU 的指令指针被恢复成 372。因为过程是否被激活只有等到程序真正运行的时候才知道，所以活动记录对应的存储区域只能动态地进行处理。栈可以用来实现这一目的，因为反激活（过程返回）的顺序和激活（过程调用）的顺序是相反的

4）现在 CPU 执行 LOG 函数的代码。这段代码将会在 AR + 3 的位置找到参数 Y，然后在 AR + 4 和 AR + 5 的位置找到两个局部变量。LOG 函数的代码将计算得到的 LOG(Y) 数值保存到为返回值预留的位置，也就是 AR 指向的位置。

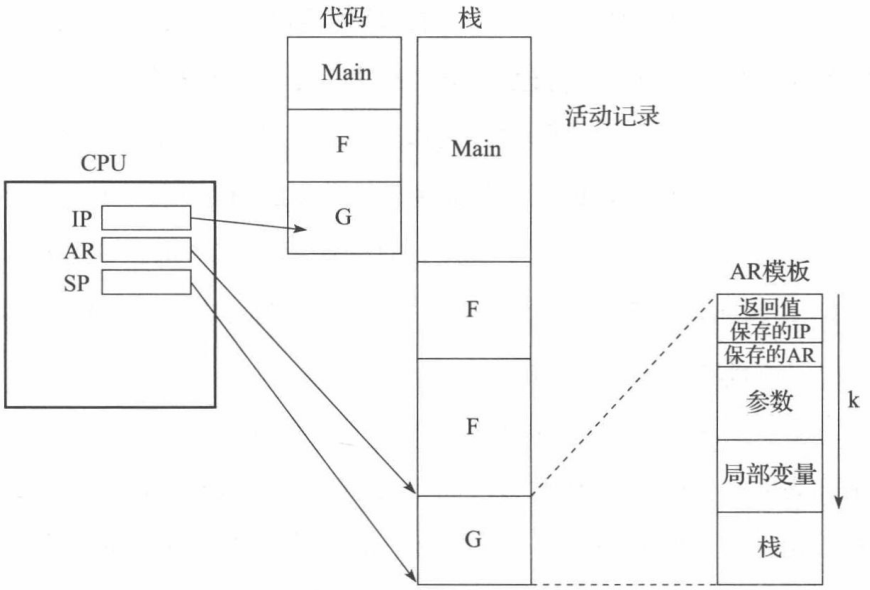


图 4.9　通过对 CPU 做一点修改，就可以把一个运行中的程序（如图 4.8）的全部活动记录存放到程序的栈中。一个过程调用发生时，它将被调用过程的活动记录压入栈顶，而当过程执行结束并返回时，它将对应的活动记录从栈顶弹出。CPU 中的活动记录（Activation Record，AR）寄存器指向当前活动记录的起始位置。在调用之前，调用者的代码会负责在栈顶建立新的活动记录，将调用参数和局部变量的数值装入栈中适当的位置。在调用发生时，IP 和 AR 寄存器被转交给被调用的过程使用，调用之前的值将在调用返回时得到恢复。当被调用的过程运行时，调用参数和局部变量都是根据它们在活动记录中的相对偏移位置来访问的。例如，第一个参数在 AR + 3 的位置。调用函数恢复运行时，它会从栈顶位置读取被调用函数计算得到的返回值

5）被调用的过程执行 RET 指令：它将 SP 寄存器指向 AR 位置，并从之前保存的位置（SP + 1, SP + 2）读取，从而将 IP 和 AR 寄存器的值恢复成调用之前的状态。现在，调用者可以从调用之处继续执行后续的指令，而这时 LOG(Y) 的值就处于栈顶。

上面所描述的过程架构是允许递归的，也就是说一个程序可以调用它们自己[5]。最早支持递归的编程语言是在 1958 年提出的 Lisp 和 Algol。设计者之所以引入递归，是因为他们希望设计一种能够表达和执行任何算法的编程语言。Lisp 语言使用 Church 的 lambda 表达式来表示算法，而 Algol 语言采用含有递归函数的过程来表示算法。1960 年左右，Edsger Dijkstra 提出以栈的形式来组织内存，同时开发出第一个 Algol 语言的编译器。相比而言，开发 Lisp 语言的编译器要困难得多，直到 20 世纪 70 年代才有了高效的 Lisp 编译器。在那之后，许多编程语言都支持递归过程。

这种过程架构的设计实际上被证明是非常有用的，不仅是在编程函数时有用，在处理计算过程中的错误时同样有用。例如，如果一个程序试图做除数为零的除法，结果会如

何？从数学意义上来说，这个结果是无定义的，这种情况下程序也无法给出一个答案。为了避免让一个无定义的值在程序中传递，CPU 的设计者让算术逻辑单元（ALU）在发生这种错误时发出一个信号。他们修改了 CPU 的指令周期，让 CPU 来检测这个信号。这就是我们在前面介绍过的指令周期中的第四个步骤。如果 ALU 检测到“除零”错误并设置相应的信号，CPU 就会转向一个特殊的子程序，这个子程序或者消除这个错误，或者终止这个程序。将 CPU 转向处理错误的子程序这一过程被称为“中断”。Organick（1973）将中断描述为一种“预料之外的过程调用”。

发生除数为零的错误，并不是使 CPU 中断的唯一原因。设计者将所有需要 CPU 立刻处理的事件统称为*异常状态*。异常状态可以分为两类：错误和外部信号。错误是指程序中可能导致不正确或者未定义行为的一种状态。能够被 CPU 检测到的错误包括：除数为零、算数上溢或下溢、缺页、保护冲突，或者任何的越界引用等。而外部信号则表明有更高优先级的事件发生。外部信号的例子包括：时钟警告、磁盘操作完成、鼠标点击以及网络数据包到达等。当任何异常发生时，CPU 都会中断当前正在执行的程序，转而去处理错误或者对外部信号进行响应。通过引入一种“中断向量”，当感应器报告异常 k 发生时，CPU 能够自动并且快速地调用第 k 个中断处理程序来进行处理（见图 4.10）。

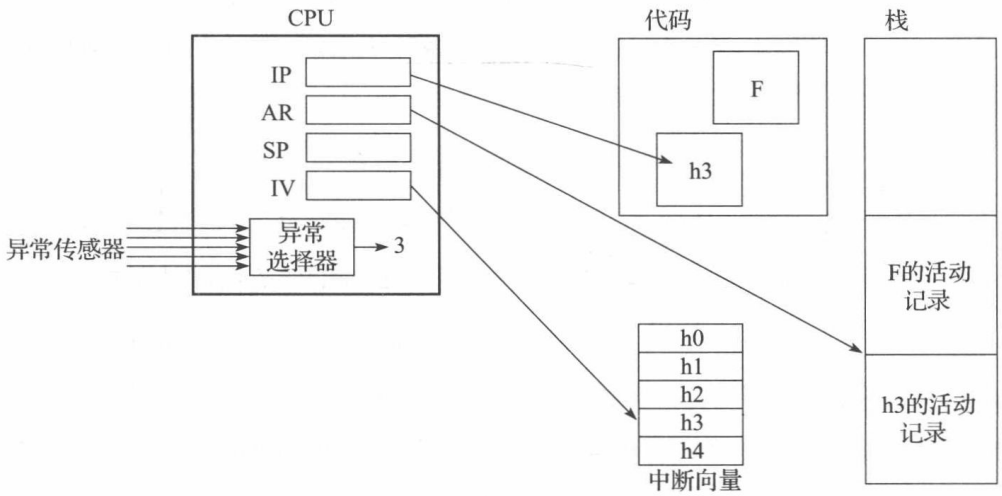


图 4.10 中断机制使得 CPU 能够将当前的任务暂停，并且执行一个进行差错处理或者响应更高优先级外部信号的过程（程序）。CPU 中的传感器电路以及计算机系统其他的传感器，都会检测可能出现的任何异常。选择电路则会将优先级最高的异常选出，并输出它的型号（如果没有异常则输出 0）。在每个指令周期的末尾，CPU 都会进行异常检测。如果存在异常，CPU 就会暂停正在执行的指令，然后以中断类型号（这里是 3）为索引在中断向量中查找。中断向量是一个存储着每个中断处理程序入口的线性表。找到中断处理程序入口之后，CPU 调用中断处理函数（图中是 h3）。h3 对应的活动记录被压入栈顶，然后指令周期恢复正常运转。中断程序执行完毕后，返回指令会让计算机恢复到之前被中断的程序位置，继续往下执行

选择的不确定性

中断机制可能会导致一种介稳态，一种难以察觉但却是毁灭性的错误。当 CPU 正在试图读取记录信号的触发电路的同时，如果出现了一个异常信号，将会发生什么？时钟可以控制的是 CPU 什么时候去查看中断信号，却无法控制外部信号到达的时机。

以下是事情的发生经过。假设在电路中采用 3 伏来表示 0 电平，采用 5 伏来表示 1 电平。外部信号的到达会导致中断触发器由 0 向 1 状态转变，也就是触发器的输出电压从 3 伏转向 5 伏。然而，这个过程需要时间，那么中间会存在一小段时间，在此时间内输出电压介于 3 伏到 5 伏之间。这个电压离 3 伏和 5 伏都不够靠近，以至于无法可靠地判断为 0 电平或者 1 电平。电子工程师称这种输出为“半电平”。半电平输入可能会导致触发器进入一个介稳态，这个状态使得输出电压为两个稳定状态的中间值。这个中间点就像是把一个球放在屋顶中间凸起的位置：它可能在这个位置停留一个无法预料的时间长度，直到出现一些微小的变化让它失去平衡状态。

介稳态给所有读取触发器输出的电路都带来了故障风险。如果这种半电平信号持续到下一个时钟节拍，后续电路接收到的信号就无法被判定为 0 或 1，在这种情况下电路的行为将变得不可预测。

介稳态问题在硬件工程师中广为人知。Chaney 和 Molnor（1973）以及 Kinniment 和 Woods（1976）对这个问题进行了实验，度量了介稳态可能出现的概率以及持续的时间。通过将计算机的时钟频率与外部信号的频率同步起来，他们试图在每次外部信号改变时都生成一个介稳态的事件。科学家通过示波器观察到大量的介稳态事件，一些事件甚至持续了 5 个、10 个甚至 20 个时钟周期（见图 4.11）。其他工程师早就知道，选择电路（有时也被称为仲裁电路）是非常难以设计的（Seitz 1980，Denning 1985，Ginosar 2003）。

77
~
78

从那时起，芯片制造商就一直在考虑电路中可能出现介稳态的问题。Sutherland 和 Ebergen（2002）在报告中表示，现代的触发器能在大概 100 皮秒（100×10^{-12} 秒）的时间内完成状态切换，而持续时间为 400 皮秒或更长的介稳态，将会每 10 小时出现一次。芯片制造商 Xilinx.com 则表示，当时钟频率为 200MHz 或更低时，他们制造的现代触发器将不会有机会出现介稳态，但是在更快的时钟频率下则会出现介稳态（Alfke 2005）。在一个每秒出现五千万次中断信号的实验中，当时钟频率为 300MHz 时，大约每分钟会观察到一次介稳态现象，而当时钟频率为 400MHz 时，每两毫秒会观察到一次介稳态

现象。而在一个每秒发生 500 次中断的计算机系统中，中断发生的频率是上述实验中的 1/100 000，可以推断出亚稳态状态出现的频率是：时钟频率为 300MHz 时每两星期出现一次，时钟频率为 400MHz 时每 3 分钟出现一次。

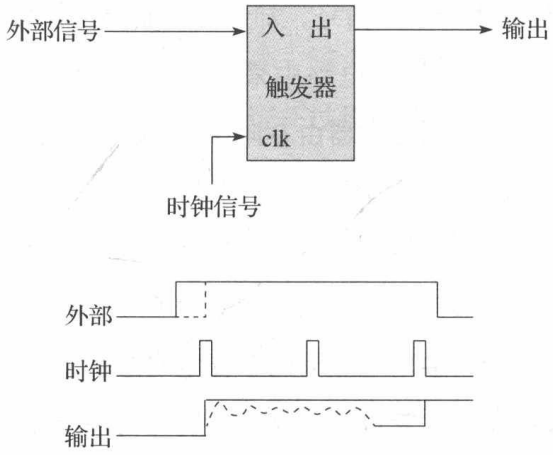


图 4.11 用来观察触发器（FF）介稳态现象的实验设置。每一个时钟脉冲信号都触动触发器的状态以匹配输入信号。当脉冲信号到达时，如果输入信号在变化（外部信号中的虚线部分），触发器可能进入一个持续数个时钟周期的不确定状态（输出信号中的虚线部分）。在数字计算机中，不确定的输出在下一个脉冲时则成为其他逻辑电路的输入，从而造成半信号故障

现在问题就出现了。当 CPU 需要查询状态时，中断触发器可能还处于介稳态，这会导致下一个控制 CPU 状态的触发器进入介稳态。如果这些触发器到下一个时钟节拍还没有进入稳定状态，那么 CPU 的行为将变得不可预测。实验结果表明这种情况很有可能会发生。

这个问题困扰了很多早期的计算机系统。在工程师真正理解这个问题之前，他们只是发现 CPU 会在一些随机的时间点停止运行，仿佛被冻结。他们甚至将这种奇怪的 CPU 冻结现象解释为“宇宙射线导致的崩溃”，因为这些现象看起来像是由于晶体管的功能遭到随机破坏而导致的。只有完全关机并重新启动，才能让这些 CPU 重新开始工作。这个问题很让人困扰，因为这种冻结现象几乎每几个小时或者每几天就会出现一次。

在 1970 年左右，英国剑桥大学计算实验室的一位硬件工程师 David Wheeler，找到了这些神秘的 CPU 冻结现象的原因：中断触发器输出的半信号。他设计了一种新型触发器，称之为“阈值触发器”，并设计了这种触发器的使用方法，消除了 CPU 在查询中断触发器状态时可能造成的停机风险（见图 4.12）。

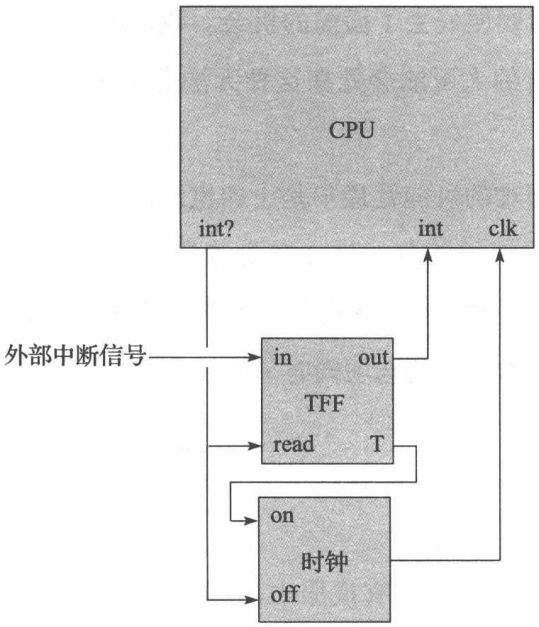


图 4.12　阈值触发器（TFF）能够保证，当 CPU 在指令周期的末尾进行中断查询时，CPU 的中断输入（“int”）是稳定的。当 CPU 请求询问外部的中断信号的值时（“int?”），它会触发 TFF 将当前的外部信号记录到它的状态中，同时把时钟关闭。当它的状态返回到“0”或者“1”时，TFF 就会通过输出 T 发送一个脉冲信号，从而恢复时钟信号。时钟只会暂停很小的一段必要时间，以确保 TFF 重新进入稳定状态。David Wheeler 是在 20 世纪 70 年代对剑桥的 CAP 计算机进行中断检查时，提出了暂时关闭时钟的这个想法

硬件工程师在设计计算机的时候，遇到了很多像上面提到的中断触发器这类问题。几乎每个部件都存在需要同时从两个事件中进行选择的现象。比如，两个 CPU 同时访问内存中同一个位置，两个交易同时锁定了数据库中的同一条记录，两台计算机同时在以太网上广播数据，两个数据包同时到达网卡，自动代理同时收到两个请求，或者机械子系统同时发现两个可以执行的选项。在上述所有情况中，如果我们要求在下一个时钟节拍到来之前在所有选项中做出选择，那么就很可能出现选择电路未进入稳态的可能性。如果我们希望等待电路进入稳态，那么就需要暂停时钟。

至此，我们可以用选择不确定性原理来总结上面这些现象：在一个预先设定的期限内，不可能无歧义地对几乎同时发生的事件做出选择[6]（Lamport 1984，Denning 2007b）。这种不确定性的来源就在于，当两个不同的信号同时发生改变时，产生的冲突可能会导致选择部件产生介稳态。

选择不确定性这种现象，其本质不是关于系统如何对观察者做出反应，而是观察者如何对系统做出反应。人类在做选择的时候也是如此。如果有很多选项同时摆在我们前面，却需要在很短的时间内做出选择，我们会怎么做呢？有时候，当截止期限到了，我们

依旧不能做出决定，这时候便失去了面前的机会。我们不能像 Wheeler 那样，暂时关闭时钟直到有了决定再说。有的人可能会处在这种无法做出选择的状态几秒、几小时、几天、几个月，甚至几年。

出现无限期的选择困难的可能性最早是十四世纪的哲学家 Jean Buridan 提出的。他描述了这样一个悖论：一只饥饿的狗，站在两份等量的食物中间，难以选择，最后饿死了（Lamport 1984，Denning 1985）。如果从当今认知科学家的角度来讨论这个问题，Buridan 可能会说，当有几个诱惑力相等的选项同时呈现在面前的时候，大脑会被固化在一个亚稳定的状态。

79 ~ 81

结论

本章的目的是通过展示各种具有说服力的细节，来表明我们完全有可能建造一种电子机器，它可以计算出任何一种人们能够找到计算方法的函数。这种机器由一个处理器（CPU）和一个存储器（RAM）构成。在一个反复执行的指令循环中，处理器执行存放在 RAM 中的一串机器指令，对同样存储在 RAM 中的数据进行操作。机器指令实现了最基本的操作，包括算术运算、内存读写以及流程控制。每一条指令都是由 CPU 中的某个电路来实现的。我们展示了当 RAM 以栈的方式存储数据时如何设计一个简单的指令集。执行基本操作、选择以及重复迭代这些指令，赋予了计算机非常强大的计算功能。

过程调用机制允许我们单独编写一些程序，并在任何其他程序中的任意位置对它进行调用。在处理异常条件时，操作系统使用过程机制来中断程序。

本章的最后讨论了选择不确定性问题，即仲裁电路在同时接收两个输入的时候有可能进入一种亚稳态，从而无法在下一个时钟节拍到来之前做出选择。这个问题源于电路的物理设计，可以通过暂时关闭时钟来解决，直到电路进入稳态。

我们对机器的上述讨论再次强调了计算过程中物理部件的核心重要性。机器中的所有指令和数据都是以二进制 0 和 1 的形式存储在电路和媒介中的。而 0 和 1 就是媒介中不同状态的表示。指令按照预先约定的方式精确地操作这些状态。而程序其实就是按照某些精确模式将计算方法中的步骤组织而成的一个指令序列。机器读取程序中的指令，并在数据上执行这些指令，所有的一切都是自动进行的。机器中的电路只是简单遵循电子和物理规律，它并不需要理解穿梭于其中的各个电信号的含义。

本章所提到的栈结构仅仅是执行程序可能的各种模型中的一种。每一种模型都有自己的规则和机器架构，但它们都完成同一件事：控制电路来读取输入并计算输出。

82

第5章
Great Principles of Computing

程序设计

对计算机真正有兴趣的人最好对底层硬件的工作原理有所了解，否则可能会设计出非常奇怪的程序。

——Donald Knuth

程序设计是应用数学最困难的分支之一，可怜的数学家们最好还是呆在理论数学的围栏里别出来。

——Edsger Dijkstra

Ada Lovelace 据说是人类历史上第一个计算机程序员。1843 年，她在“Babbage 分析机笔记”中设计了一个程序：这个程序可以让分析机计算产生一个伯努利数列。遗憾的是，Babbage 没有能够完成分析机的构造，因此，Lovelace 设计的这个程序也从来没有真正运行过。

对于程序的输入与输出数据，Babbage 计划使用提花织机时代的打孔卡片来表示它们。历史学家可能会认为那些为织布机确定卡片序列的工人也是程序员：他们为织布机的运行指定了准确的规则，而且还真实看到了这些规则的运行结果。现代程序员则会认为织布机程序员不是计算机程序员，因为织布机是无法计算数学函数的。

历史学家也可能会认为一个发明了某种算法的数学家也是程序员，例如，Newton 提出的求解联立方程组的消元法（1670），或者 Napier 提出的对数运算计算方法（1614）。现代程序员同样也很有可能不会认同这种观点，因为这些算法并不是为一个特定的机器设计的。

距 Lovelace 的程序出现近 100 年后，才出现了真正运行在电子计算机上的程序。这个时代的第一批程序员是为 ENIAC 计算机编写程序的女性们：她们在一个插线板上通过在不同的插孔之间连线来指定程序的行为（1949）（见第 1 章图 1.3）。两年之后，在 EDSAC 和 EDVAC（第一批具有“存储程序”功能的计算机）上，程序员开始在纸带上采用二进制数字的方式编写算法。在第二次世界大战期间，为军方计算弹道轨迹的女性们也是程序员，只不过她们编写的程序不是针对机器的指令，而是用于指导自己操作机械式计算机。实际上，她们的角色就是一种人力计算器。

从一开始，程序员就发现他们花费了大量的时间去发现自己编写的程序中的错误（Wilkes 1985）。即使是最优秀的程序员也无法完全避免错误的出现。于是，计算机的设计者开始设计更高级的程序设计语言和动态错误检查方法，以尽可能减少程序中错误的出现。这样的努力至今为止仍在进行。设计出没有错误的软件不仅仅是一个可靠性问题，还是一个安全性问题（包括软件自身的信息安全以及软件对环境的安全）。

本章的目的是想展示这样一个事实：虽然现实中人们发明了数量众多的计算机语言，但所有的计算机语言都具有翻译器的基础角色，即自动地将程序员书写的源程序翻译成机器可理解的指令。进一步，人们发明了一类被称为“编译器生成器”的程序：它们接收一种计算机语言的语法描述，然后输出一个针对这种语言的编译器程序。自动化的编译器生成极大地降低了计算机语言的实现成本。

程序、程序员和程序设计语言

程序是对算法的表达，并可以在一台计算机上运行。程序设计语言是一种人造语言，具有其自身的用于表达程序的语法规则。程序员是一个采用特定的程序设计语言编写程序的人：程序员需要保证他们编写的程序能够在计算机上正确运行并完成期望的工作任务[1]。

在第一种商业程序设计语言在20世纪50年代末出现之后，各种程序设计语言如雨后春笋般不断出现。在2014年，维基百科上给出了一个包含500多个程序设计语言的列表，其中的每一种语言都曾经或正在被用于商业软件的开发。如果再考虑每种语言的轻微变体、更新或升级等因素，我们可以发现数千种程序设计语言[2]。每一种语言都有其设计目标，通常是针对特定领域的问题对计算设计进行优化。数以千计的领域都涉及计算，因此，存在数千种程序设计语言也非常自然。

表5.1给出一些重要的程序设计语言。其中，对于最初出现的4种语言，有3种至今仍在使用。只有Algol语言不再被使用。但是，许多后来出现的程序设计语言都继承了Algol的语法规则。

表5.1 一些广泛使用的程序设计语言

语言	出现时间	目 的
Fortran	1957	对数学计算公式的有效处理
Algol	1958	支持递归过程的通用程序设计语言
Lisp	1958	对Church演算中lambda表达式的有效表达（第一个使用在人工智能领域的程序设计语言）
Cobol	1959	对商业、金融、管理领域计算操作的支持
APL	1962	面向数组函数处理的专用语言

（续）

语言	出现时间	目　的
JCL，Shell	1966	操作系统中的任务处理控制语言，其后继者是用于 Multics（1968）和 Unix（1972）操作系统的 Shell 语言
PL/I	1966	基于 Algol 的通用程序设计语言，包含了一些面向商业数据处理和系统程序设计的特征
Simula	1967	第一个面向对象的程序设计语言，用于离散事件仿真
Pascal	1970	支持结构化程序设计的类 Algol 语言，在程序设计教学中广泛使用
Smalltalk	1971	通用的面向对象程序设计语言，从 Simula 借鉴了消息传递机制；1980 年被标准化
Prolog	1972	对逻辑关系的有效推理（应用于人工智能、自然语言处理、定理证明等领域）
C	1973	面向系统软件的程序设计语言，一开始用于 Unix 内核的编写，后来用于 Linux 操作系统的编写
CLU	1974	基于抽象数据类型的面向对象语言
Ada	1983	美国国防部指定的一种可用于军用软件开发的程序设计语言
Sisal	1983	基于函数组合的语言，适用于具有并行处理能力的计算机
Perl	1987	广泛应用于 Web 页面设计的脚本语言
Java	1995	具有跨平台特性的面向对象语言
Python	2000	通用程序设计语言，支持函数式、命令式以及面向对象式编程，强调程序的易读性

84
~
85

Algol 语言的重要创新之一是采用形式化的方式来描述语法。这种形式化方法被称为 Backus-Naur 范式，或者简称为 BNF 范式，以其发明者 John Backus 和 Peter Naur 命名（Backus 1959，Knuth 1964）。BNF 范式能够清晰地定义一种程序设计语言的结构以及如何将这种语言结构忠实地翻译为相应的机器代码。BNF 范式也是构造编译器程序的基础。后来的研究工作使得人们可以开发出非常高效的编译器，甚至编译器生成器（即能够根据一种语言的语法规则自动生成相应的编译器）。下面我们简单介绍编译器的基本原理。

这里给出了一个 BNF 规约的片段，其中使用了赋值语句在算术表达式与数字和变量之间建立关联：

```
<assign>    ::= <var> = <A>
<A>         ::= <A><aop><A> | <M>
<M>         ::= <M><mop><M> | <F>
<F>         ::= (<A>) | x
<aop>       ::= + |–
<mop>       ::= *|/
```

其中，每一对尖括号都声明了一个语法元素：尖括号中的字符串即为该语法元素的名称。元素 <A>、<M>、<F> 分别表示加减表达式、乘除表达式、因子表达式。元素 <aop> 和 <mop> 分别表示加减操作符和乘除操作符。双冒号加等于号“ ::=”表示“被定义为”的含义。竖直线“|”表示“或”的含义。字符 x 表示任何变量名或任何数字常量（见图 5.1）。这种语法还规定了与算术运算相同的运算操作优先级，即：乘除运算的优先级要高于加减运算。在后面的一节中，我们会展示一个编译器如何使用这些语法树去生成相应的机器代码。

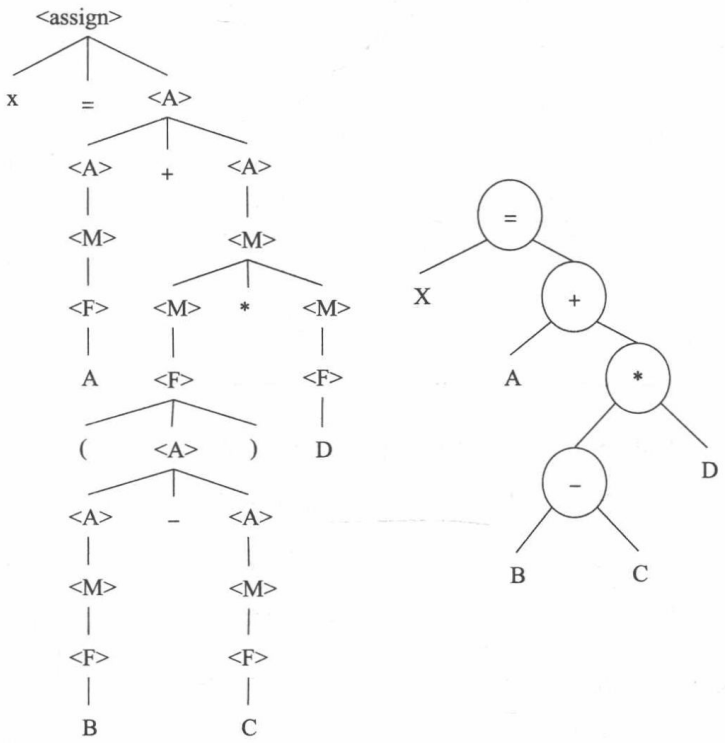


图 5.1 赋值语句 X = A + (B − C)*D 可以被两种类型的语法树所表达。一种是图中左边所示的语法树：它基于文字的语法去解释这个表达式的语法。另一种是图中右侧所示的更简洁的语法树，其中仅仅保留了操作符和操作数。在结构确定之后，原始语句中的括号没有必要存在，因为这种表示方法具有与算术运算相同的运算符优先级

程序设计实践

程序设计是一种编写在计算机上运行的指令序列的实践活动。程序员的工作包括两个主要部分：

1）针对一个问题，设计相应的可以在计算机上运行的解决方案（即程序）。

2）验证一个程序在计算机上的运行结果是否正确地解决了所要解决的问题。

在程序设计中，程序员可以使用工具来辅助其工作。其中，设计相关的工具包括程序设计语言和图形化编译器（图形化编译器可以确保程序员设计的程序完全符合程序设计语言的语法规则）。编译器、链接器和解释器等工具可以将一个源程序转换为相应的机器指令序列。测试和调试工具可以帮助程序员定位并修正错误。函数库和版本控制工具可以保证程序的最新性以及对程序的有效复用。在使用这些工具的过程中，程序员就像是一个传统的技术工匠。

不同级别的程序员通常具有不同的程序设计技能水平。一个新手程序员通常会花费大量的时间来理解程序设计语言的语法以及一些基础的算法。中级程序员能够在无需指导的情况下完成相当数量的标准程序设计任务。高级程序员能够进行大型软件系统的设计与实现，并能够在系统的不同抽象层次上灵活迁移。程序员的生产力（例如可以通过代码行数进行度量）通常具有非常大的差异：很多优秀的程序员可以达到中级程序员 10 倍以上的生产力，一些非常优异的程序员甚至可以达到 50 倍。

86
~
87

一些程序员已经成为传奇式的人物，因为他们编写的程序对世界产生了重大的影响。John MacCormick（2012）列举出了 9 个著名的程序算法，其设计者包括 Len Adleman、Sergey Brin、Jim Gray、Tom Mitchell、Larry Page、Ron Rivest、Adi Shamir 等人。由这些伟大的程序员所提出的一些设计思想已经成为计算的基本原理。

许多软件系统因其巨大的影响力也被视为传奇。例如，VM/370、Multics、Unix、Windows 以及 MacOS 等操作系统；Oracle 数据库系统；Akamai 页面缓存系统；TCP/IP 协议簇；域名系统（DNS）；被出版社使用的数字对象定位符（DOI）。这些系统的一些开发者也因此成为传奇人物，包括 Vint Cerf、Fernando Corbato、Bill Gates、Bill Joy、Bob Kahn、Alan Kay、Butler Lampson、Paul Mockapetris、Roger Needham、Jon Postel、Rick Rashid、Dennis Ritchie、Jerry Saltzer、Ken Thompson 等。

大型的软件系统通常包括数量众多的不同软件程序，需要数百甚至数千程序员的群体协同才能实现。操作系统和微软的办公软件应用 Office 是这种大型软件系统的典型实例。如何组织数量众多的程序员协同工作以达到生产力的最大化、错误率的最小化以及软件的按期交付，绝不仅仅依赖于天才型的程序员。这个问题对项目的协同管理和测试提出了重大的挑战。这种挑战到目前为止还在困扰着软件产业。Fred Brooks（20 世纪 60 年代 IBM 360 操作系统项目的管理者）将他对大型软件项目组织和管理的很多深刻见解都记录在《The Mythical Man Month》（Brooks 1995）这部著作中。在第 10 章中，我们将对大型软件系统设计的基本原则进行探讨。

程序中的错误

自从20世纪40年代软件出现以来，在人们的印象中，软件总是非常容易出错。程序员总是被程序错综复杂的结构弄得头晕脑涨，并且不得不花费大量的时间去消除自己编写的程序中的错误，去不断地改进程序以使得程序能够适应花样繁多的各种外部异常情况（见图5.2）。

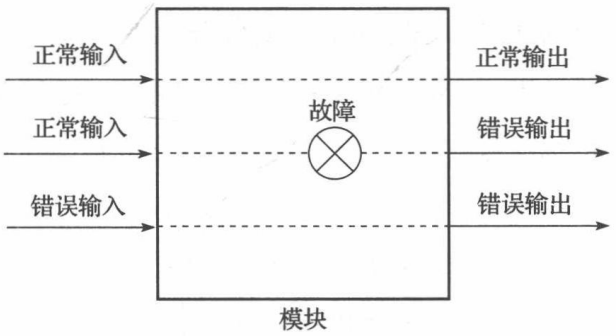


图5.2 一个软件模块的输出错误通常由两种情况引起：一种是由软件模块中存在的故障（缺陷）引起；另一种是由错误的输入数据所引起。发现程序中的缺陷或检测错误的输入数据通常非常复杂

大多数的物理系统遵守连续性法则，它则保证了一个自变量的微小变化在因变量上也仅仅会导致一个微小的变化。因此，物理系统对微小错误具有天然的容忍能力。很多生
88 物系统（包括人类和动物的免疫系统）具有自我检测和修复的机能——通过反馈和修正来应对错误。

与之相反，由软件创造出来的虚拟世界则对错误具有高度的灵敏性。程序中一个二进制位上内容的改变（从0变为1，或从1变为0）都有可能对程序的行为和运行结果带来巨大的影响。而且，人们可以非常容易地编写出与物理定律相冲突的软件，这些软件在与外部世界交互时就会导致错误的发生。消除或减少程序中的错误不是一件简单的事情，它涉及从用户期望、设计者意图、源程序到机器代码的复杂变换过程中的各个环节（见图5.3）。

对消除错误的渴望促使程序设计语言的设计者在语言中加入了各种各样的消错结构。经过多年的发展，目前比较成熟的消错结构包括类型、子程序、模块、异常、对象、包、语法编辑器、调试器，以及试图缩减程序语法与问题域语义之间距离的新型语言。虽然已经取得了很大的进展，但如何消除程序中的错误对编程者而言仍然是一个巨大的挑战。

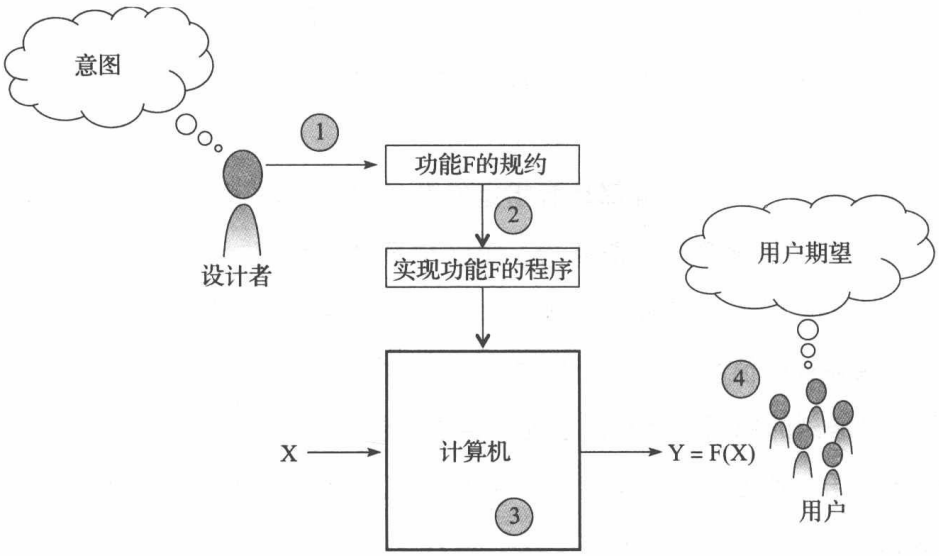


图 5.3 在将设计者的意图翻译为机器可执行程序的过程中非常容易发生错误。具体而言，有 4 种导致错误发生的原因。（1）一个程序的规约没有准确地反映设计者的意图或错误地反映了用户的期望。（2）程序员在程序中引入了错误（例如，程序员对程序的运行环境做出了不够准确的假设），或者编译器自身具有错误或被恶意植入了特洛伊木马（这将会导致源程序和机器代码之间不再具有等价关系）。（3）执行程序的计算机存在由各种因素导致的错误、缺陷或故障。（4）用户对计算机的期望行为与计算机的实际行为之间存在不一致。自动验证器试图消除由第 2 种原因导致的错误。容错系统设计方法试图消除由第 3 种原因导致的错误。基于原型的用户反馈机制则试图消除由其他两种原因导致的错误

对消除错误的渴望促使软件工程师（更加专业化的程序员）采用了传统的 4 阶段工程设计过程：（1）需求，（2）规约，（3）原型，（4）测试。在需求阶段，工程师通过与用户或问题域中的其他利益相关者的交流，确认当前开发的软件系统所有需要或不需要的行为。在规约阶段，工程师采用特定的技术语言对满足用户需求所必需的各种功能进行形式化的准确描述。在原型阶段，工程师实现一个可运行的软件系统原型。在测试阶段，工程师对软件原型进行各种各样的测试，检查该原型能否表现出规约中规定的软件行为或能否满足用户的预期。在这个过程的实际执行中，工程师会根据最新的信息对相关的阶段和活动进行不断地迭代。例如，在撰写规约的过程中，工程师可能会发现一个需求具有二义性；这时，他们必须要与用户进行交互，以解决这个二义性问题。在实现原型的过程中，工程师可能会发现某个规约可能会导致严重的效率问题，从而返回对规约或原始需求进行调整。在测试过程中，工程师可能会发现一个性能问题或一个错误，从而返回对实现代码进行检查、对规约进行审查或与用户进行必要的交互。对于包含众多模块、涉及大规模开

发者的大型软件系统而言，这样的工作流程是非常复杂的；因此，通常会使用专业的决策追踪系统、协同支持系统以及版本控制系统等软件工具对整个开发过程及制品进行系统化的管理。

89
~
90

即使一个软件项目的开发者都具有丰富的开发经验，软件项目的实施进度和成本也通常会远远超出预期和预算。一般认为，由于预算严重超支、交付时间严重延迟或重要功能的缺失等原因，60% 的软件开发项目都不能算是成功的；30% 的软件开发项目在没有开发完成前就会被取消[3]。虽然项目失败的原因通常指向管理者对时间和成本的错误估算，但其背后更深刻的原因则是软件系统的异常复杂性，这种复杂性远远超过了任何一个经验丰富的软件开发者的能力范围。可以看到，寻求有效的软件设计基本原理也具有非常重要的经济原因。

自动翻译

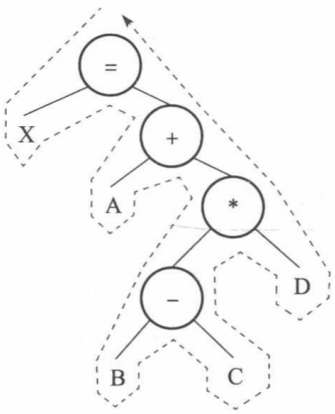
程序设计的一个重要部分是如何将程序员编写的源程序转换 / 翻译为机器可执行的程序。目前存在两种方式来实现这种翻译：编译器和解释器。编译器是这样一个程序：它接受源程序作为其输入数据，根据程序设计语言的语法规则解析形成相应的语法树，最后生成相应的机器代码作为输出。解释器也是一个程序：它也对输入程序进行语法解析，但是并不会生成对应的机器代码，而是根据解析的结果实时调用相应的系统操作，系统操作是一个已经被编译为机器代码的子程序。经过多年的持续发展，这两种自动翻译方式之间逐渐失去了清晰的边界。两者之间最显著的一个差别可能是编译器只需要对程序进行一次翻译即可以生成多次运行的机器代码，而解释器则是在每次程序执行的过程中都需要进行实时的翻译。

下面我们来了解一下编译器的工作流程。编译器的工作目标是将一个源程序翻译为严格对应的机器指令序列（即，不能遗漏源程序声明的行为，也不能增加任何新的行为）。为了实现这一目标，我们需要两个基本条件。首先，这个程序设计语言的语法（以 BNF 范式的方式表示）必须不能存在二义性，即：对于一段合法的源程序字符串只能存在一个唯一的语法树与其对应。第二，在翻译过程中，一个语法树只能被影射到一个唯一的机器指令序列上[4]。

为简化起见，我们假设基于 BNF 语法已经解析形成了一个简单的语法树：其中，每个叶子节点代表一个常量或一个变量名，每个中间节点则代表一种操作（图 5.1 中即展示了这样一种语法树）。我们还假设这个程序设计语言支持第 4 章中提及的 4 种基本

结构：赋值、顺序、分支、循环。基于这四种结构具有完整的表达能力，可以用它们来编写任何一个算法；如果能对这四种结构进行翻译，那么就能对任何一个源程序进行翻译。

图 5.4 到图 5.7 分别展示了编译器如何将赋值语句、顺序语句、分支语句、循环语句翻译为机器代码的过程。图中左侧展示了每种语句结构对应的语法树，图中右侧则展示了实现左侧语法树的一个机器指令模版。 91




```
LA X
LA A
L
LA B
L
LA C
L
SUB
LA D
L
MUL
ADD
ST
```

图 5.4 该图展示了编译器将图 5.1 中所示的赋值语句 X = A + (B − C)*D 翻译为机器代码的过程。编译器首先将这个赋值语句解析为一个中间节点为操作符、叶子节点为操作数的语法树。然后，编译器对这棵树进行一次逆时针遍历（见图中虚线），并且按照遍历顺序将其遇到的每个叶子节点或操作符输出到一个指令栈中。输出结果（见图中右侧）是一个可以在栈机器上正确求解该赋值语句的指令序列

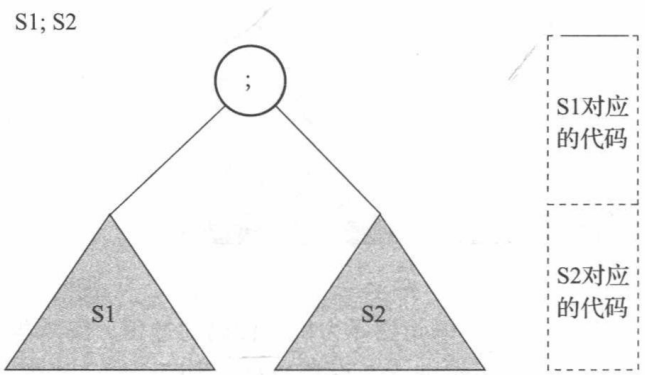


图 5.5 该图展示了编译器将两个顺序执行的语句翻译为机器代码的过程。编译器首先分别将这两个语句解析为对应的语法树，然后将这两个语法树通过一个顺序操作符（用分号表示）进行组合形成一颗更大的语法树。接下来，编译器对这棵语法树进行逆时针遍历从而形成图右侧的代码模式，即：将 S1 对应的代码放在 S2 对应的代码之前 92

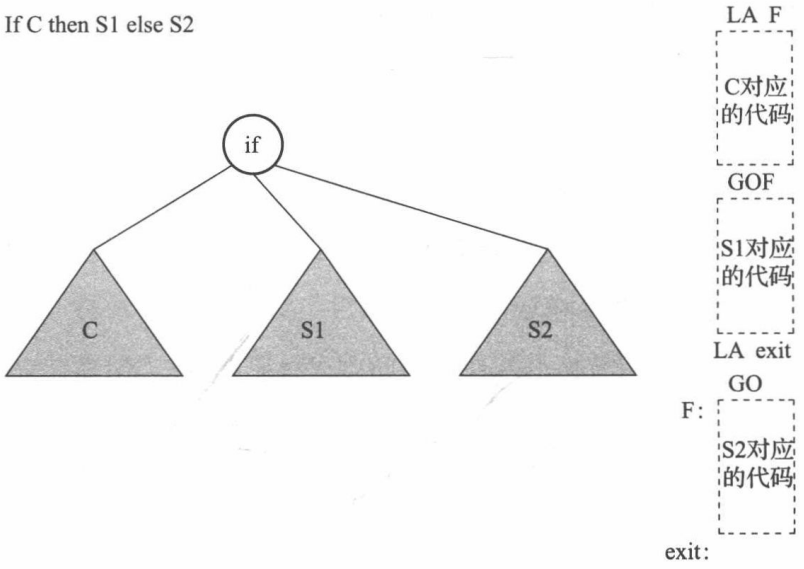


图 5.6 该图展示了编译器将一个分支语句翻译为机器代码的过程。编译器首先将分支语句的三个构成成分（条件表达式 C、语句块 S1 和 S2）解析为对应的语法树，并通过“if”操作符将这三颗语法树进行连接。然后，编译器对连接形成的语法树进行逆时针遍历，生成图右侧所示的代码模式。在每一个代码块之后，编译器插入了一个跳转语句用于保证只有当 C 为真时才执行代码块 S1 且只有当 C 为假时才执行代码块 S2。指令 GOF（Go On False，为假时跳转）假设其执行时栈中存在一个布尔变量，该布尔变量的取值由指令“LA F”和 C 中的代码所决定

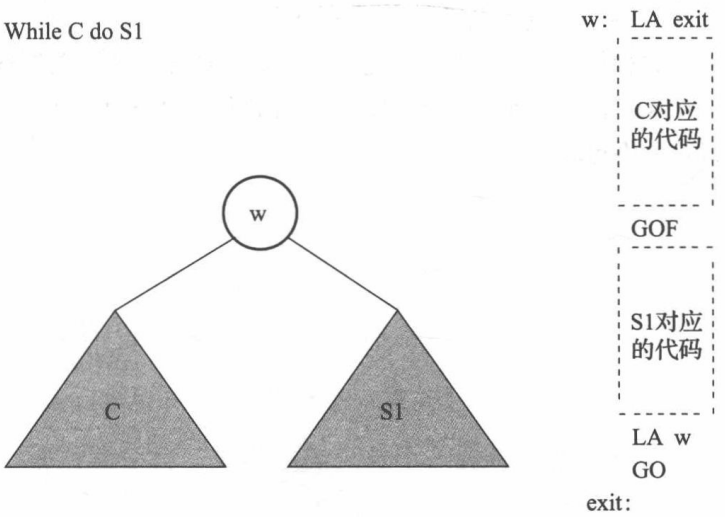


图 5.7 该图展示了编译器将一个循环语句翻译为机器代码的过程。编译器首先将循环语句的两个构成成分（条件表达式 C、语句块 S1）解析为对应的语法树，并通过“while”操作符将这两颗语法树进行连接。然后，编译器对连接形成的语法树进行逆时针遍历，生成图右侧所示的代码模式。在每一个代码块之后，编译器插入了一个跳转语句用于保证在开始执行代码块 S1 时 C 必须为真且一旦 C 为假就会跳过代码块 S1

将树转换为指令的方法称为“树遍历”。我们可以按照一个逆时针的路径（见图 5.4 中虚线路径）对树中的所有节点进行遍历；每一个叶子节点会被访问一次，而每一个 k 元操作符则会被访问 $k+1$ 次；对于每一个最近访问到的节点，都会输出相应的指令。

这种编译方法有一个缺点。它可能会导致在源程序和机器代码的表现形式上存在很大的距离。这种距离给编译器的编写者留下了更多的解释空间，有可能会导致与程序语言设计者的意图发生偏差。因此，这也增加了在代码中存在错误的可能性[5]。

为了缩小这种表现形式上的距离，程序语言设计者开始对目标机器的指令集合进行扩展。这些扩展的指令被实现为独立编译和验证的程序。因此，编译器可以直接将源程序中的一些操作直接翻译为对应的扩展指令，从而减少了可能存在的翻译错误（见图 5.8）。

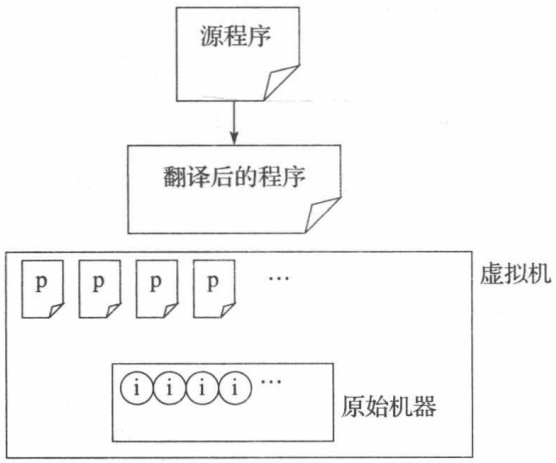


图 5.8 虚拟机是程序执行的一种基本模式。图中给出了一台带有特定指令集合的原始机器。虚拟机则是对这台原始机器的扩展，即在原始机器指令集合的基础上，形成了一组粒度更大的扩展指令（图中标示为 p 的节点）。因此，翻译器可以将源程序语法树上的操作节点直接对应到虚拟机中相应的扩展指令上。Java 语言就是基于虚拟机的方式进行程序执行的。Java 翻译器将源程序翻译为“字节码”（bytecode），即：一个能在 Java 虚拟机（JVM）上执行的代码序列。对每一种不同的操作系统（即原始机器），都需要开发各自的 JVM 实现，即：在当前操作系统指令集合的基础上构建 JVM 支持的扩展指令集合。通过 JVM 的支持，Java 语言具有了良好的可移植性

Java 虚拟机（JVM）是这种扩展指令方法的典型代表。JVM 的指令（称为“字节码”）实现了 Java 对象上的各种操作。Java 编译器直接将 Java 程序翻译为对应的字节码，而不是翻译到更底层的原始机器指令上。JVM 的指令集合实际上就是一组粒度更大的可调用程序过程，这些过程已经事先被编译为原始机器指令序列。这种两层的设计使得 Java 具有很好的可移植性。对于任何一个操作系统，都可以开发相应的 JVM，将 Java 的字节码指令集翻译为当前操作系统所支持的原始指令。因此，由 Java 编译器输出的字节码程序可以在

任何操作系统的 JVM 上执行。任何一个用户都可以在自己的机器上编写和编译一个 Java 程序，且编译后的 Java 程序可以在无需任何改动的情况下在任何具有 JVM 的机器上运行。

操作系统也是虚拟机的一个典型代表。操作系统管理各种类型的资源，包括进程、虚拟内存、消息通道、文件、输入－输出、目录等。对每一种类型的资源，操作系统都有一个子系统对其进行管理。例如，文件系统通过一组可以施加在文件上的标准操作对文件进行管理，具体包括创建、删除、打开、关闭、读、写 6 种操作。文件系统虚拟机通过对外提供这 6 种扩展操作，极大提高了文件管理的可靠性和安全性。在操作系统的运行过程中，文件系统的这 6 种操作被封装为系统调用，避免了通过第三方函数库对文件进行管理的过程中可能存在的不可靠因素。

操作系统中存在一个被称为“ shell”的构成成分，其实现了一个任务控制语言。用户通过与 shell 的交互，告诉操作系统他们想要执行什么程序以及程序的输出和输出放在那里。shell 将用户的一条命令解析为一棵语法树，并基于语法树来决定进行什么系统调用（见图 5.9）。不同于编译器，shell 对每一条用户命令都进行一次实时解析。这也是程序解释执行的一个典型实例。

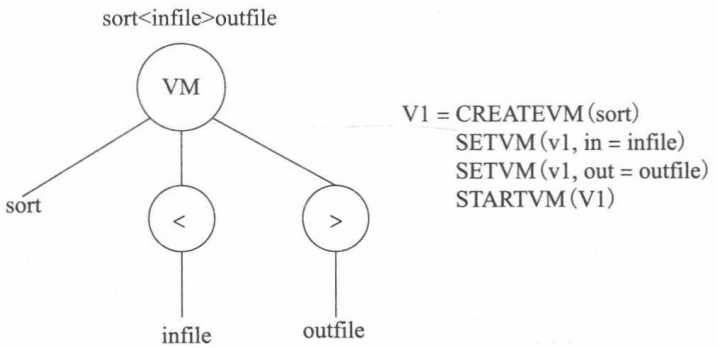


图 5.9 Unix 的 shell 语言可以让用户输入一个程序的名字，例如 sort。sort 程序从用户指定的输入文件“infile”中读取数据，然后将程序运行结果输出到用户指定的输出文件“outfile”中。用户在系统中输入这些信息（如图中左上角所示）。符号“<”表示下面即将出现的是输入文件的名字。符号“>”表示下面即将出现的是输出文件的名字。解析器将这行信息解析为如图中左下方所示的语法树。通过对这棵语法树的遍历，shell 解释器生成了如图中右侧所示的 4 个系统调用。第一个系统调用创建了一个可以运行 sort 程序的虚拟机。第二个和第三个分别指定了这个虚拟机的输入和输出文件。最后一个系统调用触发了虚拟机的执行。当虚拟机执行完毕后（即 sort 程序执行结束），用户输入的这一条命令也执行结束，然后用户可以再次输入一个新的命令

总结

程序设计是一种为特定的问题提供计算化解决方案的活动。好的程序设计是一种通

过专业训练和长期实践而形成的技能型活动。

程序设计语言具有领域特定性，它使得编程者设计特定领域问题的计算化解决方案的过程更加便利。由于程序设计语言通过 BNF 范式进行语法定义且人们都知道如何构造针对 BNF 语法的解析器，因此，对于任何一个新的程序设计语言而言，生成相应的编译器是一件非常容易的事情。程序设计语言越来越强的领域特定性使得编程者的意图和语言的表达能力之间的语义距离越来越小。

程序中的错误是让编程者非常恼怒的一件事情。在为特定问题提供计算化解决方案的过程中，有 4 个关键因素会导致错误的发生：能否准确地表达利益相关者的意图和期望、能否准确地进行程序设计、能否准确地编译程序、能否准确地执行程序。这里，“准确”一词的含义是指与设计者的原始意图一致。系统设计者已经发明了一系列的重要技术，来帮助程序员避免错误，进而编写出可靠、可用、安全的程序。

从源程序到机器代码的翻译过程可以完全自动化地进行。只要这种自动化翻译过程经过有效的验证，编程者就可以充分相信一段机器代码程序会精确地表现出其源程序所要表达的行为。为了更好地理解，我们简单介绍了编译器将 4 种标准程序结构（赋值、顺序、分支、循环）翻译为机器代码的基本过程。

当源程序与机器代码在表达形式上存在较大的语义距离时，语言设计者和编译器设计者不得不开始考虑如何应对在自动化翻译过程中可能存在的错误。编译器设计者使用的机器代码模式在程序设计者看来可能没有正确地表现出源程序设计者的意图。减少这种错误的一种方式是通过虚拟机提供粒度更大的指令集合，从而缩小源程序和编译后程序在表达形式上的差异。虚拟机所支持的扩展指令，由于粒度更大，可以与源程序中的操作建立直接的对应关系。例如，文件系统提供的创建、删除、打开、关闭、读、写 6 种操作实际上就是作用于文件的一组虚拟指令。如果可以将文件输入输出相关的源程序直接翻译为这 6 种虚拟指令（而不是翻译到更底层的机器指令上），那么，编译器的设计可以得到极大的简化。

编译器是进行程序翻译的一种模式。另一种模式则是解释器：它解析一条语句，并在解析过程中动态地对语法树上的操作进行调用。许多语言（例如，逻辑语言、列表处理语言、命令语言）在解释执行模式下会具有更高的运行效率。一个重要的原因是很多操作涉及对内存的动态使用，而一个动态的运行时系统能够快速地适应这种动态性（比如通过垃圾内存回收机制实现对内存的动态管理）。

通过使用这些技术，我们可以更加充分地确认机器代码确实完全精确地表达了源程序所要表达的行为。如果这一点无法得到有效的保证，程序设计语言的使用效果就会大打折扣。

第 6 章
Great Principles of Computing

计　　算

信息高速公路，就是不带重量的比特以光速在全球移动。

——Nicolas Negroponte（1996）

这句话是错的。

——Alfred Tarski

计算就是对一组比特不断进行改变的过程，每一次改变只花一点点时间和能量，仅仅影响到很少的几个比特。

计算的工作量大小是通过完成这项计算任务所需的时间（或能量、空间）来衡量的。那么，一台计算机去完成某项任务到底需要多少时间（能量、空间）呢？我们是否能够预测某一项任务或者某一类相关任务的计算量呢？

自 20 世纪 30 年代以来，这些关于性能方面的问题就给计算任务的设计者们带来了持续的挑战。他们提出了下面这四个问题：

1）这个任务存在一个解决算法吗？

2）这个算法需要运行多长时间？

3）是否存在更快的算法？

4）若有，更快的算法是什么？

这些问题激励着我们不断去探索、去理解计算的复杂度，也就是计算一个可计算函数的值并得出正确结果所需要的步数。表 6.1 总结了我们发现的四类主要的函数，这些类别将指引本章后续的讨论。

这个表中提到了判定问题，也就是判定一件事情是真（1）还是假（0）。计算机科学家用判定问题去评测各种计算问题的困难程度。这么一种明显的限定却不会给我们的讨论带来任何损失。对于任意一个函数 $F(x)$，都存在一个对应的判定函数 $\mathrm{DF}(x, y)$，使得当 $F(x) = y$ 成立时它返回 1，不成立时返回 0。这个判定函数的计算复杂度并不高于一般函数，有时候判定函数要比一般函数更简单。但是，如果判定函数计算起来很困难，那原始函数也一定不会比这更简单。

“容易”和“困难”这个术语是相对的。“容易”意味着我们在计算机上花费一段合

理的时间就能对大量样本做出判定；“困难”意味着我们在一段合理的时间内只能对很有限的样本做出判定。非常困难，或者说难解，则意味着即便是使用世界上最快的计算机，也要运行很多个世纪，才能对我们所关心的样本做出判定。而在容易与非常困难之间，是超过 3000 多个来自各种领域的实际问题（技术上被称为 NP 完全问题）。这些问题本身是难解的，但验证答案却比较简单。如果有人能找到其中某一个问题的快速算法，那么所有其他问题也就将迎刃而解了。至于这样的一个快速算法是否存在，被认为是数学和计算领域最前沿的开放性问题之一。

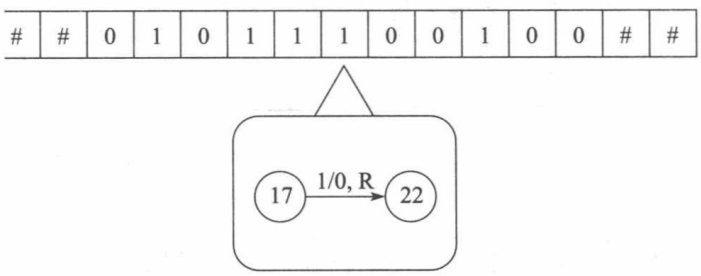
表 6.1　按计算难度区分的几类计算问题

类　型	描　述	专业名称	难度级
容易	存在已知快速算法的判定问题	多项式的（Polynomial，P）、易解的	$O(\log n)$ 和 $O(n^k)$，k 为常数
困难，但容易验证	不存在已知快速算法，但可以快速进行验证的判定问题；通常涉及在一个大数据集上进行搜索的过程	非确定性多项式的（Nondeterministic Polynomial，NP）、NP 完全的	$O(2^n)$、$O(n!)$ 或更差
非常困难	不存在已知的快速算法或快速验证算法的判定问题	难解的	$O(2^{f(n)})$，其中 $f(n)$ 是指数级或更差
不可计算	不存在任何解决算法的判定问题	不可计算或不可判定的	

在上述表格中，困难程度的级别是通过解决问题所需要的步数随着问题的输入规模 (n) 而增长的速度来描述的。$O(n)$ 代表计算时间随着输入规模的增加而呈现线性增长，$O(2^n)$ 意味着计算时间随着输入规模的增加呈现指数式增长。这些困难度级别的具体含义在后续会详细阐述。 100

图灵机

图灵机（1937）一直以来都是公认的计算参考模型，它战胜了很多与之竞争的其他模型，例如 Post 的重写系统、Church 的 λ 演算以及 Gödel 的递归函数。图灵机之所以如此成功，要归因于它与真正的自动计算机最相似，并且可以计算任何其他模型所能计算的任何函数。



图灵机包含一个有限状态的控制单元，可以在一个无限长的二维纸带上移动。纸带上的每个小方格含有一个符号，可以是0、1或者#，其中#代表空格。控制单元被定义为形如（q，a/b，LR，q'）的多元组集合，其中q和q'是控制状态，a和b是纸带上的符号，LR是一个方向指示。例如多元组（17，1/0，R，22）表示“如果处于状态17并且当前方格里的符号是1，那么向这个方格里写入0，向右移动一个方格并进入状态22”。还有一个状态是停机状态，如果机器进入这个状态，它就会停止，而此时纸带上的内容就是输出。

图灵还描述了一个可以通过多元组来模拟任何其他机器的通用机器。通用图灵机引发了Church-Turing命题（猜测）：任何可计算的模型都能用图灵机来表示。到目前为止还没有发现任何新的计算模型打破这个猜想。当然，有一些函数是图灵机无法计算的，例如停机判定问题。而其他的一些函数，例如图像标记（图3.7），可通过人机
101 协作来计算，但目前为止还没发现图灵意义上的可计算方法。

图灵机是一种很方便的计算复杂度参考模型。一个算法的复杂度，可以通过在给定输入条件下图灵机运行完该算法所需要的移动次数来衡量。本质上按串行方式执行的图灵机，非常适合计算移动次数来判定一台机器从开始状态运行到停机状态所需时间的上界。

验证器（verifier）这个概念对应于一种“非确定性的”图灵机。在这样的机器里，对于一个特定的机器状态和当前纸带符号，可能有不止一个对应的多元组。例如，机器可能既有（17，1/1，L，24）又有上面提到的多元组（17，1/0，R，22）。那机器应当使用哪个多元组？机器需要做出选择，因为仅凭当前状态和纸带符号无法唯一确定接下来的状态。非确定性机器会计算这一系列的可选项，看看是否其中有某一系列选项能导致停机并得到特定的输出。通过把程序的每个可能选择构造成一棵树，非确定性自动机可以转化为确定性自动机，但这棵树可能会很快变得庞大无比。

验证器仅仅是确认一个非确定性自动机中有一条计算路径可以得到一个特定的输出。这要比枚举所有可能性并依次检查要容易得多。

现代计算复杂性理论不再直接应用图灵机，但仍然依赖于程序的精确定义，以及程序计算一个函数所需要的步数。

简单问题

我们先来看一些简单计算的实例以及运行时间的度量标准。

实例 1　简单的线性搜索

找到本书中首次出现“图灵”这个词的页面号。一个解决之道是用一个字符串来表示这本书（字符串中嵌入了表示页边界的标记），然后开始搜索“图灵”这个词，沿着这本书的字符串一个字符一个字符地滑动，每次滑动都检查当前字符是否能与“图灵”匹配。如果我们找到了一个匹配的字符，可以立刻停止算法并输出当前位置所在页面的页号。

检查要搜索的字符串与书中的每一个子串是否匹配，它所需的时间与要搜索的字符串的长度呈固定比例关系。不妨假设待搜索的字符串长为 s，我们需要对书中每一个字符对应的子串进行匹配检查。如果这本书共有 b 个字符，那么总共的搜索时间不会超过 $An + B$，其中 A、B 是常量且 $n = sb$，我们称这种时间为“ n 级”，因为 n 较大时，时间与 n 成比例关系，简记为 $O(n)$，这种搜索方法称为线性搜索。 102

实例 2　二分搜索

简单的线性搜索可能不会是最快的。假设这本书有一张索引表，表上记载了每个单词首次出现的页码。基于索引表，我们可以进行二分搜索：重复询问我们要搜索的字符串是位于当前查询的索引表的上半部分还是下半部分。在每一个阶段，我们将搜索域缩小到上个阶段搜索域的一半。这意味着我们需要 $\log_2 n$ 次划分就能把搜索域的规模降到 1，最终检查该单词是否是我们要搜索的字符串（见图 6.1）。

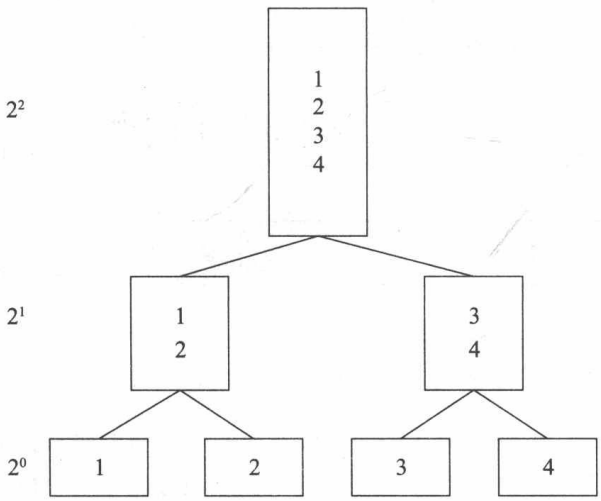


图 6.1　二分原理就是将一个大规模的集合，通过一系列的划分（每次一分为二），分解成很多只具有单个元素的小集合。例如这里的集合 {1，2，3，4} 先被对半划分，每一半再进行划分。每一层标有 0、1、2，其中 0 在树的最底层。如果 $n = 2^k$，则树的深度为 $k = \log_2 n$，这个对数可看作是以某个基底（这里是 2）进行划分直至单个元素所需要的划分次数。很多计算问题都可以用这种“分治”策略进行求解，因此这种对数符号经常出现在计算任务的时间复杂度级别记号中

因此，在索引表中找到字符串“图灵”所需的时间正比于$A\log_2 n + B$，其中A和B为常数。我们称这种时间为“$\log n$”级，因为n较大时，时间与$\log n$成比例关系，简记为$O(\log n)$。二分搜索要比线性搜索快很多，例如一本约有2^{20}(大约一百万）个字符的书，线性搜索需要2^{20}级别的时间，相较而言二分搜索只需要20级别的时间，整整快了50 000倍。对于大文件来说这个优点更为明显。我们下一步要关注的就是建立这样一张索引表需要多少工作量。

103

实例3 排序

建立索引表就是一种排序的实例。排序是指把一系列的数据项按某个顺序排列起来。例如，一系列单词可以按字母顺序排序，一系列数字可以按递增或递减排序。有些排序可能会有重复的元素，基于索引的书则对排序做了点小扩展，还提供了指针来指出每个单词在书中的原始出现位置。

排序的实现方法之一是逐次最大值法（successive maxima)。这种方法扫描包含n项的整个序列，找到其中最大的元素，并与最后一个元素进行位置交换，然后对剩下的$n-1$项重复这个过程，再对剩下的$n-2$项进行重复，依此类推。这种做法一共需要比较$n(n+1)/2$次，复杂度为n^2级，或记为$O(n^2)$（见图6.2）。

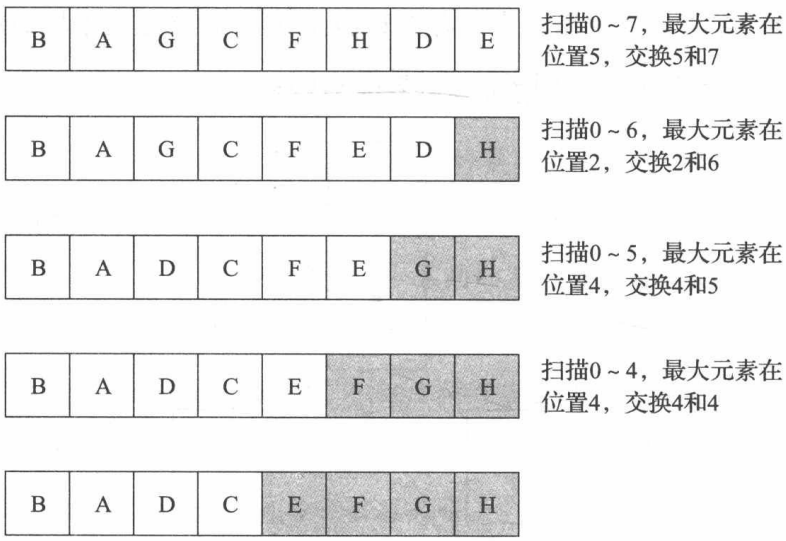


图6.2 通过逐次最大值法，可将一个无序的序列变成一个有序的序列。这里是一个在0 ～ 7位置共有8项元素的待排序数组。第一遍（第一行），通过检查每个位置找到最大的元素——位置5的“H”，将位置5和位置7的元素进行交换，从而让最大值出现在位置7。第二遍（第二行），对0 ～ 6位置的这些项重复上述过程，然后第三遍对0 ～ 5位置的项重复，以此类推。当执行完只有0、1两个位置的这一遍后算法就终止，比较的总次数为$n(n+1)/2$次。对于有n个正数的情况比较发生的总次数为$n(n+1)/2$次，本例中是36次，这种方法的复杂度为$O(n^2)$

还有更快的排序算法吗？有，不过对于初学者来说可能不是那么显而易见（见图 6.3）。更快的算法要应用二分原理，把原始序列划分成两部分，而每个部分继续被不断划分后进行排序，这需要重复 log$n$ 次划分（见图 6.3）。而在每一层，全部的 n 项都会被检查，总共时间为 $O(n\log n)$。

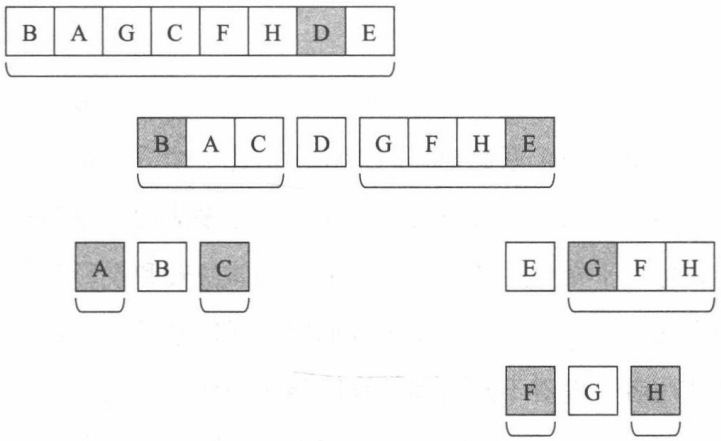


图 6.3　快速排序算法把待排序列对半划分，使得左边的每个元素都小于右边的任一元素。为了达到这个目的，第一遍，我们随机选择一个元素（D），然后把小于等于 D 的元素都移动到左边，比 D 大的都移动到右边（见第二行）。现在 D 的左边有一个长度为 3 的待排序列，右边是一个长度为 4 的待排序列。对左边的待排序列采用同样的划分做法，B 是我们随机选到的元素。对于右边也是同样，E 是随机选到的元素，如此重复下去，直到只有一个元素。在实际操作中，上级序列有指向下级子序列的指针，用于把一个元素移动到左边或右边。因此当只有一个元素的子序列开始合并时，元素已经处于正确的位置了。如果每次随机选择的元素都能把原序列分成两个，那么快速排序算法一共需要 $\log_2 n$ 遍，每遍将检查所有元素，因此总时间为 $O(n\log_2 n)$

我们能做得更好吗？不能。想象一下有一个完美的算法，它至少要包含一个过程 F 来告诉我们，原始序列的每一项在排好序的序列中的位置。$F(i)$ 表示第 i 项在排好序的序列中的位置，F 需要产生比特流来代表在排好序的序列中的位置。如果序列一共有 n 项，则每个位置字符串的长度至少为 log$n$，因此 F 需要时间 $O(\log n)$ 来产生该字符串。对 n 项中的每一项都需要进行这个过程，因此整个序列就需要 $O(n\log n)$ 的时间。

从这些例子中我们能够看出什么？

- 首先，有些问题可以比其他问题更快地解决。二分搜索比线性搜索或排序要更快。
- 第二，有时如何找到更快的算法并不是那么显而易见的，甚至是否存在更快的算法都不确定。发现逐次最大值排序法很容易，而发现快速排序算法（QuickSort）却并

不容易。甚至 $O(n\log n)$ 事实上是最快的时间级同样也不是一件容易发现的事。

- 第三，设计者一般会进行权衡。例如当搜索发生得比较频繁时，设计者往往倾向于愿意承担构建索引表的成本，从而使得二分搜索得以实施。

104 上述例子包含了 $O(\log n)$、$O(n)$ 和 $O(n^2)$，下面这个例子是一个 $O(n^3)$ 级的算法。

实例 4 矩阵乘法

一个矩阵可看作是由 n 行、n 列，总共 n^2 个元素组成的方形数组。在线性代数中，我们通常用一个未知向量和一个系数矩阵的乘积来描述一个方程组，并用矩阵乘积的形式来表达方程组的解。矩阵乘法是非常重要的操作，在很多应用中也经常被使用。最典型的一种应用是图形学，包括平板电脑和智能手机上的物体旋转和投影都是通过矩阵相乘来实现。软件执行矩阵乘法是如此之快，以至于我们完全无所察觉，我们只能“看”到屏幕上的物体旋转。尽管使用者一般看不到具体的矩阵乘法操作，因为它们通常被封装在软件库里，但矩阵相乘无处不在。按照线性代数书上的标准方法去计算两个 $n \times n$ 矩阵的乘法，其常见时间复杂度为 $O(n^3)$（见图 6.4）。

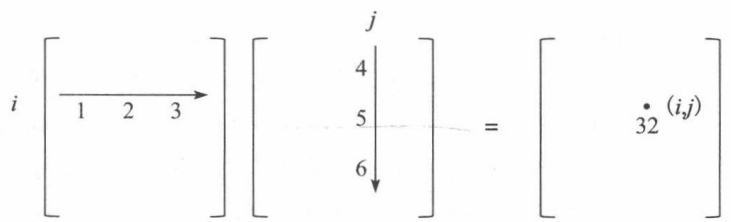


图 6.4 方阵相乘算法把两个大小为 $n \times n$ 的方阵相乘，得到一个同样大小的矩阵。计算结果中，位置 (i, j) 上的值的标准计算方式是“将第一个矩阵的第 i 行与第二个矩阵的第 j 列进行向量相乘”。向量相乘定义为对应位置数值乘积的加和。在本例中，$n = 3$，图中所示的行与列相乘为 $(1, 2, 3) \cdot (4, 5, 6) = (1 \cdot 4) + (2 \cdot 5) + (3 \cdot 6) = 4 + 10 + 18 = 32$，这称为“点乘”。每次点乘操作需要执行 n 次乘法和 $n - 1$ 次加法操作，共计 $2n - 1$ 次操作，而这样的点乘操作需要重复进行 n^2 次，因为需要对结果矩阵中的每个元素都进行一次点乘计算，因此总计算量为 $n^2(2n - 1) = 2n^3 - n^2$，也就是 $O(n^3)$

我们刚刚看到的这四个例子是已经知道快速算法的经典问题实例。它们的计算量都是 $O(n^k)$ 级，k 为常数。$O(\log n)$ 也包含在其中，因为它 $O(n)$ 比级还要快。

计算机科学家把所有能够在 $O(n^k)$（k 为常数）级时间内解决的问题统称为“多项式”类问题，因为其计算量可以用最高阶为 n^k 的多项式来测量。计算机科学家们约定多项式

105 类问题属于“容易”一类，因为它们在大多数的计算机上都可以很好地运行。然而还有很
~ 多问题的求解算法运行起来是很困难的，接下来我们将讨论这部分的问题。
106

指数级困难问题

任何需要枚举的问题都可能会是很困难的。想象一下，假如你需要检测状态域中的每一种状态看其是否满足一定特性，你可以把所有的状态当作一个长长的序列并依次检查，所需的时间将会与状态域的大小成正比。当状态域很大时，算法将需要运行很长一段时间。

实例 5　所有的十位数字

打印所有的十位数字看起来似乎很容易。因为一共有 10^{10} 个数，做这件事的算法将会是

$$\text{For i} = 0, \cdots, 10^{10} - 1, \text{print i}$$

算法的时间复杂度为 $O(10^{10})$。这到底意味着多长时间呢？对于一个现代高速芯片，时钟频率为 1GHz（每次执行 10^9 次操作），大概需要 10 秒钟。我们又需要多少张纸呢？假使我们用很小的字体打印，每页分成五栏，每栏 100 个数，一页也就是 500 个，若是双面打印那么一张纸可以打印 1000 个数字，因此我们需要 $10^{10}/10^3 = 10^7$ 张纸。一个含有 20 个装纸盒的箱子一般能容 10 000 张纸，我们需要 $10^7/10^4 = 10^3$ 个纸箱。倘若每个箱子大 107
小为 3 立方英尺[㊀]，那么我们需要一个 10×30 平方英尺[㊁]占地面积，10 英尺[㊂]高的储藏室，一张桌子上都放不下这些打印出来的东西。

即使我们真有这么一个储藏室，更糟的问题还在后面。高速打印机一般每秒可打印一张纸，整个工作将会持续 10^7 秒，大约是 4 个月之久。

问题的难度并不在于算法本身，而在于枚举所有状态所需的时间。从第 3 章中 10 的幂指数表中，我们可以观察到即使一个很小的指数都会产生一个非常巨大的数字。任何需要指数级运行时间的算法都会是很困难的。

实例 6　背包问题

背包问题是调度研究领域的一个经典问题[2]。我们有一个给定大小的背包，以及 n 个物品。这些物品的大小分别是 $\{s_1, \cdots, s_n\}$，每个物品对应的价值分别是 $\{v_1, \cdots, v_n\}$。我们希望找到能够放进背包的具有最大价值的一个物品子集。

背包问题是很多实际问题的抽象模型，例如一个旅行者该把哪些物品放进有限大小

㊀ 1 立方英尺≈ 0.0283 立方米。
㊁ 1 平方英尺≈ 0.0929 平方米。
㊂ 1 英尺≈ 0.3048 米。

的背包里，再例如一个货运公司要决定如何让货车载重更多，或者一个装配线管理员如何安排任务，使得既可以按时完工又能获得最大利润。

当 n 很大时，背包问题会变得很困难，因为有非常多种可能的子集——事实上会有 2^n 个子集（见图 6.5）。如果将它们一一枚举并计算大小，然后选出最优者，这会花费很多时间，因为这样的搜索过程需要 $O(2^n)$ 的时间。例如有 10 个物品就会对应有大约 1000 个子集，20 个物品会有大约 100 万个子集，30 个物品将会有 10 亿个子集。对于这种问题，还没有人发现更好的算法，不过目前也没有人证明不存在更快的算法。

指数式的搜索有可能通过启发式方法来避免。对于背包问题，一个常用的启发式方法是计算每个物品的“价值 / 大小”比率，然后按照价值 / 大小比率逐渐降低的次序把物品依次放入背包，直到无法放进更多物品为止。排序的存在决定了搜索是线性的，且总的运行时间为 $O(n\log n)$。不幸的是，这种启发式方法不能保证找到的解是最优的解（见图 6.5）。

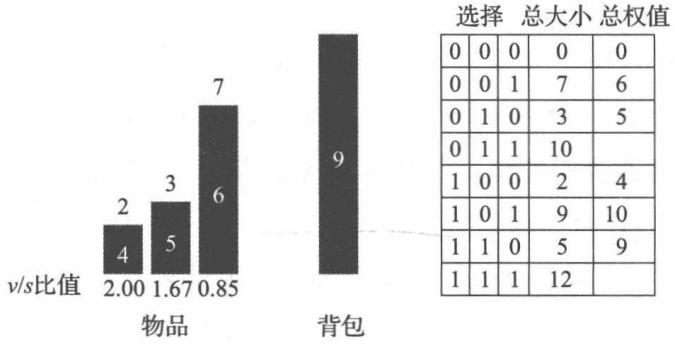


图 6.5 背包问题致力于对于给定大小的背包和 n 个物品，找到能够放入背包的价值最大的物品子集。例如本例中，我们有三个物品大小分别是 {2, 3, 7}，对应的价值是 {4, 5, 6}，背包大小为 9。右边的表格枚举了这三个物品组成的 8 个子集的情况，1 表示在子集中，0 表示不在。右边两列分别表示该子集中的物品大小总和以及子集的有效价值（当子集大小不超过 9 时）。可以看出包含物品 {1, 3} 的子集是最终选择的方案。一种启发式的做法是用价值 / 大小的比率去标记每个物品，然后按照比率降序把物品依次放入背包，直到再也放不进去更多物品为止。这种策略达到的解是 {1, 2}，接近于真正的最优解，但不是最优解

实例 7 访问所有城市

每个货运公司都关注这样一个问题：在 n 个城市之间运送货物的最短路线是什么？
108 这是一个经典的旅行商问题。我们可以通过搜索来寻找最短路线：首先，枚举这些城市之间所有可能的路线，然后计算每条路线的长度，其中最短的那条路线就是问题的解。

上述方法需要花费多长时间？可以通过先选择第一个城市（有 n 个选择），再选择第二个城市（有 $n-1$ 个选择），直到全部选择完，来构造每一条路线，一共有 $n! = n(n-1)(n-2)\cdots 2\cdot 1$ 条路线，这种方法具有 $O(n!)$ 级的复杂度。当 n 比较大时，$n!$ 的 Sterling 近似是 $n^n e^{-n}\sqrt{2\pi n}$，即使对于一个较小的 n 而言这都是一个巨大的天文数字，例如 100！就高达 10^{158} 级别。如果我们有一个时钟频率为 10GHz 的处理器，每个时钟内可以处理一条路线，那么每秒可以处理 10^{10} 条路线，一年可以处理 10^{17} 条路线（一年包含 3.14×10^7 秒），一个世纪可以处理 10^{19} 条路线，整个的计算过程将需要 10^{139} 个世纪。相比较而言，一个星球的寿命一般只有 10^7 个世纪。所以这个问题很难解！

对于背包问题，启发式方法能非常有效地工作。2004 年，瑞典的一个工作小组通过启发式来寻找瑞典的 24 978 个城市之间的最优路线，他们的计算过程花费了分布于 96 个处理器上的总共长达约一个世纪的 CPU 时间。[3]

109

实例 8　合数分解

合数是两个或多个质数的乘积。在密码学领域，很多重要的算法都依赖于由两个质数生成的合数。例如 RSA 加密系统，使用两个大质数作为私有密钥，而对应的合数作为生成公共密钥的基础（Rivest et al. 1978）。这种公钥是安全的，因为目前还没有快速算法能把一个合数分解成两个质数。所有已知的整数分解算法都需要在 $2^{n/2}$ 种可能的分解上进行搜索，这具有指数级复杂度。RSA 是目前所知的唯一能够确认是安全的公钥加密系统。然而，如果存在快速的合数分解算法，就能够从公钥中快速地提取出两个质数，RSA 加密系统也就失效了。

以当前的硬件水平和分解算法，2056 位的公钥是安全的，但到 2020 年它可能就会被破解了，4096 位公钥则被认为是永远不会被破解的。1994 年，Peter Shor 发明了一种能在多项式时间内进行合数分解的量子算法。如果能够建造出量子计算机，那么 RSA 系统可能就没有用武之地了。

这些例子表明只有当我们找到一个无需枚举和搜索整个可能空间的算法，我们才能使分解合数变得容易，才能完成这个原本运行到世界毁灭都无法完成的事情。

根本性的原因在于搜索空间的指数级（或更糟糕的）增长。在第 2 章中我们给出了 1000 的指数表以及它们的前缀名，包括 giga（G）、tera(T)、peta(P)、exa(E) 及 zetta(Z)。但我们的命名无法跟上数据量级的增长，如今每年网络上的数据量为 1Z，而不久之后数据总量就会超过 1Y(yatto)，这是我们仅剩的一个前缀。

由此我们可以看出，$O(2^n)$ 或 $O(n!)$ 级的问题和任何多项式问题相比计算起来都更困难，因此我们称之为指数级困难问题。

从实际来看，即使对于一个很小的输入，这些问题也都是无法求解的。唯一的办法就是利用启发式方法，搜索整个可能空间中的一个很小的子集来近似求解。我们都知道启发式方法可以很快地给出结果，但无法确定这个结果离最优解偏离有多远。瑞典的那个小组如此幸运，他们的启发式方法找到了一条能被证明是最优的旅行商路线。

110 不幸的是，唯一已知能够求解指数级困难问题的算法就是枚举法。这样的问题如此常见而普遍，研究人员研究了许多年试图找到快速算法，可惜至今都未能成功。

计算困难但容易验证的问题

指数级难判定的问题有一个有趣的特征：一个方案可以被很快地验证。把背包问题表述为下面的这样一个判定问题：是否存在价值至少为 k 的方案？这种判定形式并不比原问题容易。然而，验证一个提出的方案是否具有少于 k 的价值只需要线性的时间，因为只需要把所选物品的大小和价值加起来即可。旅行商问题也类似，验证所提的一个路线长度是否超过 k 也只需线性的时间。

我们已经看到计算机科学家们引入记号“P”来代表能够在多项式时间内解决的问题集合，这里 P 代表多项式级（Polynomial）。任何 P 类的问题都可以在 $O(n^k)$ 时间内解决，其中 k 为一个常数。

计算机科学家引入记号“NP”来代表可以在多项式时间内验证的问题集合，这里 NP 代表非确定性多项式级（Nondeterministic Polynomial）。非确定性是一种技术表达，指可以猜测到正确答案并在多项式时间内验证。对于 NP 问题，所有已知的算法都需要指数级或者更糟的运行时间，但对于每一个方案我们可以在 $O(n^k)$ 时间内进行验证。

为了讨论 NP 问题，我们需要使用一些记号。对于任何一个 NP 问题 A，有一个具有多项式运行时间的验证算法 V_A，$V_A(x, y)$ 验证 $A(x) = y$ 是否成立。$V_A(x, y)$ 是一个判定问题，因为它的输出只能是 0 或 1（见图 6.6）。

我们容易注意到，验证器可以用来寻找问题 A 的解。假设我们想针对某一个给定的 x 计算 $A(x)$，并且已知问题的解最多包含 n 位比特。我们可以枚举所有的二进制数 $y = 0, 1, \cdots, 2^n-1$ 并计算 $V_A(x, y)$，即验证是否 $A(x) = y$。如果其中任何一个验证通过，那我们就找到了 $A(x)$ 的解。这个搜索过程是 $O(2^n)$ 复杂度的，因此是指数级困难的。直接求解 $A(x)$ 的算法可能会比这样搜索验证要快，验证搜索不一定是最好的解决方法。

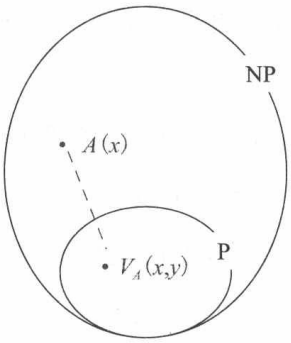


图 6.6　这个韦恩图表明了 P（多项式）类判定问题与 NP（非确定性多项式）类判定问题之间的关系。每一个属于 NP 类的判定问题 $A(x)$ 都有一个属于 P 类的验证器 $V_A(x, y)$ 用来回答“$A(x) = y$?”是否成立。很多大家熟知的指数级困难问题都是 NP 问题，因为我们可以很快地对一个假定答案进行验证。不过暂时还不知道是否存在能够快速解决 NP 问题的算法。这就是 P = NP 问题，数学和计算科学领域中最重要的未解决问题之一

NP 完全

NP 问题可以在多项式时间内被验证，这点燃了探索多项式时间内解决 NP 问题的 111
希望。我们很快将会看到，如果某个特定 NP 类中的某一个问题能在多项式时间内求解，那么所有其他的 NP 问题也都可以在多项式时间内求解，这将使得 P = NP。不过至今无法证明 P 是否等于 NP。这个问题也是数学和计算科学领域的最重要的未解决问题之一（Cormen et al 2009）。

科学家和工程师们在面对新问题时，经常将它们“归约”成一个以前的问题，而这个以前的问题已经有良好的解决办法。例如，电子工程师使用通用电路模拟器来判断他们设计的电路是否合理，通过专门的描述语言向模拟器描述这个新的电路要比单独为这个新电路做一个专门的模拟器要更容易。

算法研究者也使用同样的方式去探索解决问题的不同算法之间的关系。假设我们可以把方法 *A* 能解决的一个问题实例转化为方法 *B* 也能解决的形式，那我们就可以用 *B* 来得到 *A* 的解。这就是从 *A* 到 *B* 的一个归约。这样一个归约表明目标方法 *B* 至少和方法 *A* 一样强大。

当然需要强调的是，归约必须是很快的——多项式时间级的。如果归约转化过程需要 112
花费指数级时间，那使用 *B* 将没有任何意义，因为这无法给我们探索 *A* 的多项式级算法带来任何希望。

如果我们可以找到这样一个 NP 问题 *B*，使得每一个 NP 问题 *A* 都可以在多项式时间内归约到 *B*，那将会怎样？这个问题的解将会适用于每一个 NP 问题，使我们探索 NP 问

题的快速解法变得容易：如果我们找到这个基本问题 B 的快速算法，就能用它快速解决每一个 NP 问题 A。

1971 年，Steve Cook 引入了术语“NP 完全”来指代 NP 中的这种基本问题。他指出如果存在多个这样的基本问题，则每个都可以归约到其他的基本问题。只要其中的任何一个问题找到快速算法，那么所有其他的 NP 问题也就都有了快速算法。

Steve Cook 提出的第一个 NP 完全问题是电路可满足性问题（CSAT）。一个简单布尔电路可看作是一组相互连接且不包含回路的逻辑门电路（与门、或门、非门）。它的输入是一个 0 和 1 的集合 x，输出是 0 或 1。CPU 电路就是一个具有多个输出的复杂布尔电路。

假定 C 是具有 n 条输入线的一个电路，这些输入的设置可以用一个 n 比特二进制数 x 表示，$C(x)$ 的取值为 0 或 1。CSAT 问题就是：是否存在某个输入 x 使得 $C(x) = 1$？

显然 CSAT 是一个 NP 问题，因为对于给定的输入 x 我们可以很快地验证输出是否为 1，只需在电路中跟踪信号即可。然而，已知的解决可满足性问题的最好算法需要枚举所有可能的 x 并依次测试，这需要 $O(2^n)$ 的复杂度。

Cook 正是通过验证器 V_A 来将 NP 中的任何一个问题 A 归约到 CSAT。当 $A(x) = y$ 成立，验证器 $V_A(x, y)$ 的取值为 1。下面是归约转化的主要过程。

对验证器 $V_A(x, y)$ 生成对应的电路 $CV_A(x, y)$，这个电路通过一系列步骤来模拟 CPU 的操作，每个步骤可以执行验证程序的一条指令。每个步骤都会把指令执行前的所有机器状态（程序、内存、CPU 状态字、x、y）映射为指令执行后的状态，这种映射是通过整个 CPU 电路的一个拷贝来实现的。因为验证算法在 $O(n^k)$ 步完成，其中 $n = \text{size}(x) + \text{size}(y)$，对应的电路也将会有 $O(n^k)$ 个步骤。这个电路异常复杂，它必须记录内存和 CPU 状态的每一位比特来作为步骤转移之间传递的信号，而且步骤的数目也十分巨大。

一旦有了 $V_A(x, y)$ 模拟电路，经过略微修改，把 x 固定为一个给定的输入，只保留 y
113 为未指定变量，我们可以将其转换成一个电路 $C_x(y)$。如果 $C_x(y)$ 是可满足的，那么存在一个 y 使得 $C_x(y) = 1$，而这只有在 $A(x) = y$ 成立时才能满足。因此，CSAT 对于该电路的解就是使得 $A(x) = y$ 成立的 y 值（见图 6.7）。

因为 CSAT 本身是 NP 问题，并且任何其他的 NP 问题都可以归约到它，因此 CSAT 是 NP 完全的。

在 Cook 提出 CSAT 是 NP 完全问题之后不久，Richard Karp(1972) 发现了其他的 21 个 NP 完全问题。几年后，Garey 和 Johnson(1979) 收集了科学、工程、商业和其他领域共 3000 个常见问题，这些都被证明是 NP 完全的。这项工作具有很重大的意义。如果谁

能为这 3000 个问题中的任何一个找到快速算法，那该算法就可以转化为能够解决其他 NP 完全问题的快速算法，也同样可以解决其他所有的 NP 问题。大量我们暂时无法解决的实际问题到时候都有可能得到解决。这里是部分 NP 完全问题：

114

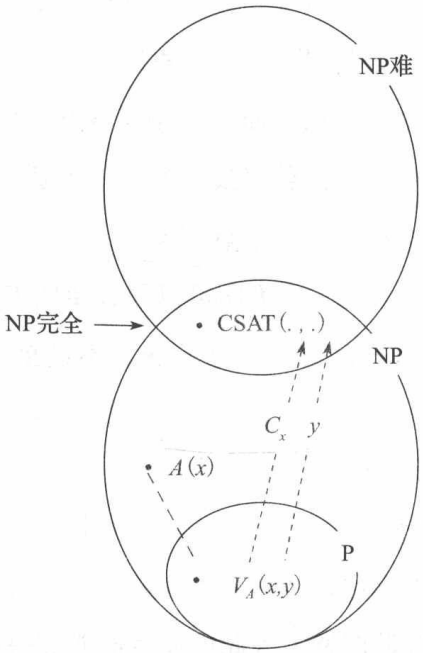


图 6.7　一个可以被任何 NP 问题归约的问题称为是 NP 难的。电路可满足性问题（CSAT）就是一个 NP 难问题，当然也是一个 NP 问题。CSAT 试图回答“是否存在某个输入使得给定的电路输出 1？”通过构造一个电路 $[C_x(y)]$ 来模拟验证器 V_A 在固定输入 x 和可变输入 y 下的行为，任何 NP 问题 A 都可以归约到 CSAT。如果 CSAT 判定“y 使电路 $C_x(y)$ 满足”，那么 $y = A(x)$ 也就得到验证。CSAT 是第一个 NP 完全问题的例子

- 电路可满足性问题（CSAT，如前文所述）。是否存在一种输入使得电路输出为 1？
- 可满足性问题（SAT，布尔可满足性）。是否存在一组输入值使得给定的布尔表达式为真？
- 背包问题。是否存在一种背包方案使得所有放入的物品其价值之和至少为 V？
- 子集和问题。给定一个包含很多数字的集合，是否存在一个非空子集使其总和为 K？
- 整理单词。对于一系列随机信件，里面包含有哪些单词？
- 哈密顿路径。给定一个图，是否存在一条路径能经过图中每个节点恰好一次？
- 旅行商问题。访问地图上所有城市的最短路径是什么？
- 团（clique）问题。在一个社交网络中，找到由彼此相互相识的节点构成的集合。
- 死锁问题。在一个网络中找出这样的节点集合，集合中的每个节点都停下来等待集合中其他节点的信号。

- 图和网络流问题：
  - 着色问题。最少需要几种颜色才能使得图中任意相邻的两个节点都有不一样的颜色？
  - 节点覆盖。找到一个节点最少的集合，使其可以覆盖图中的每条边。
  - 子图同构。给定的图是否与另一个图具有同构的子图？
  - 独立集。在一个网络中，找出彼此不相连的节点（顶点）集合。
  - 覆盖集。在一个网络中，找出数目最小的节点（顶点）集合，使得网络中不在集合中的节点都与在集合中的某个节点相连。

很多问题都被表述为图问题，这是因为图可以很方便地表示我们日常所接触的网络。

尽管很多智者都试图探索这些问题的快速算法，不过至今还无人成功。许许多多的经验、观察似乎表明 NP 完全问题根本不存在快速算法，也就是 $P \neq NP$。

Fortnow（2009, 2013）曾设想如果我们能够为某一个 NP 完全问题找到快速算法（即证明 $P = NP$），世界将会发生怎样的变化。他预测这将会促使一场经济变革，因为许多难
115 题将变得可以解决。也有很多人不这么认为，因为启发式方法上的巨大进步已经能够对这些实际问题产生可以接受的良好近似解，因此即使我们能够得到最优解，也不会有太大的进步。

不可计算问题

19 世纪 30 年代以来，研究计算的数学家们都纷纷意识到，存在一些函数是无法计算的。其原因是没有足够多的程序来计算所有的函数，从更专业的角度来说就是，所有可能的程序是可数的，而所有可能出现的函数却是不可数的：函数空间的无限性比程序空间的无限性具有更高的阶。图 6.8 简要说明了为何如此。

如果不可计算问题对于我们的日常生活来说无关紧要，那我们对它们也不会有什么兴趣。但事实上，这些问题确实具有重要影响。有许多我们想要知道的事情确实是无法计算的。图灵在 1937 年提出的一个例子就是停机问题（halt problem）：通过检查一个给定的程序和对应的输入，我们是否能够判断该程序在此输入下是否会停机（终止）。通常情况下，程序是否会终止并不是显而易见的，因为使得循环能够终止的条件可能是事先无法知道的。不终止的程序可能会一直循环下去。不管我们设计什么方法来判断程序是否终止，都不能依赖于模拟程序的运行，因为我们要评估的这个程序可能会陷入无限循环，从而模拟过程也将一直运行下去而无法得出任何结论。我们特别希望找到的是一种测试方法，它

可以在执行程序之前（即无需执行程序）就告诉我们程序中是否存在无限循环。然而，这样的测试方法并不存在。

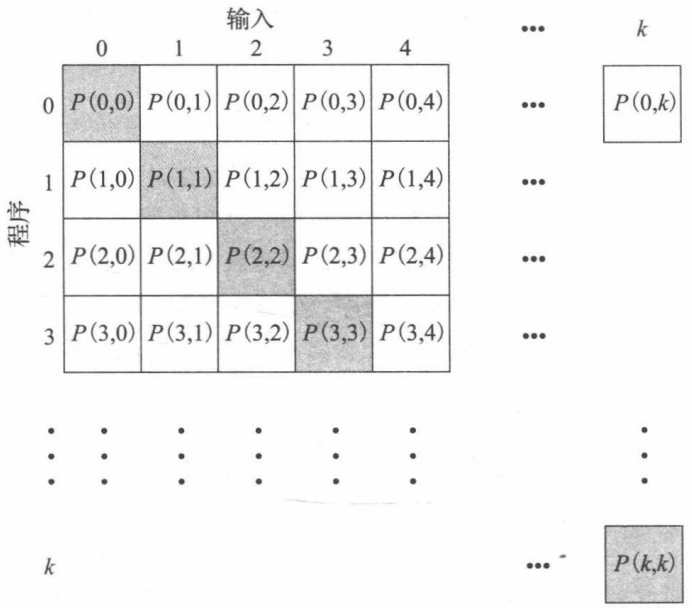


图 6.8　对角论证表明了为什么相比于所有可能的函数，程序要少得多。在顶部，我们枚举出程序所有可能的输入，每一个整数代表提供给程序的一种可能的二进制字符串输入。同样，我们可以在左侧竖向枚举出所有可能的程序，不过这需要一个更精细的过程：我们生成每一个可能的数字的对应二进制字符串，并用编译器测试该字符串是否代表了一个合法的程序；如果是，就把它添加到程序列表中。然后，我们用 $P(i, j)$ 来代表程序 i 在输入为 j 时的输出。因为每一个程序和每一个输入都会出现在上述矩阵的某处，因此该矩阵囊括了所有的程序和输入。这时，如果我们把所有的对角元素替换掉——例如把 $P(k, k)$ 替换为 $P(k, k) + 1$——这一串对角元素就定义了一个新的程序，但这个新的程序却不在上述程序列表中，因为它与已经在列表中的任何一个程序都至少在某一个输出上存在冲突。这种形式的论证被数学家 Georg Cantor（康托）用来证明存在比有理数更多的实数，因此有时候被称为“康托对角化”

图灵证明了停机问题是一个定义清楚但不可计算的问题。而图灵提供的证明是对逻辑的一种巧妙运用。他采用反证法，先假设存在一个程序 H 能够检测程序 P 是否终止：若程序 $P(x)$ 终止则 $H(P, x)$ 输出 1，若程序 $P(x)$ 不终止则 $H(P, x)$ 输出 0。我们必须假定程序 P 和输入 x 具有适当的二进制代码表示。如果存在这样的检测程序 H，我们把它作为程序 G 的一个子程序，如图 6.9 所示。这个时候，如果我们把程序 G 自身的一份代码拷贝作为程序 G 的输入，就会出现一个悖论。当输入是程序自身时，无论程序 G 行为如何，我们都会得出矛盾。因此，我们在前面做的关于存在检测程序 H 的假设是不成立的。

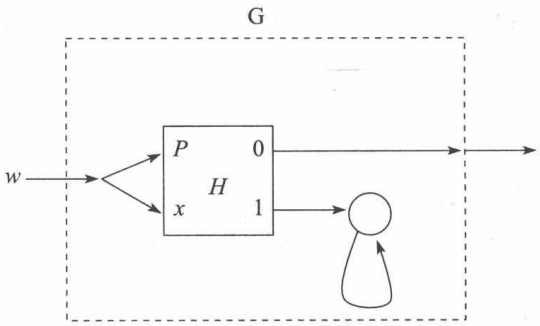


图 6.9 如果停机问题可解，则会存在一个可计算的程序 H 可以告诉我们一个程序 $P(x)$ 会终止 (1) 还是不会 (0)。因为它是可计算的，我们可以编写该方法对应的一个软件模块，然后构造一个更大的模块 G，它把 H 作为一个子程序。G 首先询问 $H(w, w)$ 是否会终止，如果回答是“会终止”(1)，G 就进入一个无限循环；如果回答是“不会终止”(0)，G 则直接把 H 的输出 (0) 进行输出。当 $w = G$，即输入是 G 的一个副本时，就会出现矛盾。如果假设 $G(G)$ 会终止，那么 $H(G, G) = 1$，这会迫使 G 陷入无限循环。如果我们假设 $G(G)$ 不会终止，那么 $H(G, G) = 0$，此时 G 却终止了。两种假设——$G(G)$ 会终止或 $G(G)$ 不会终止——都导致了矛盾。因此，我们关于 H 是可计算的假设是不正确的

这个矛盾看似如此怪异！很多人都认为这个证明令人难以置信，对此的反应是认为这是一个骗人的把戏。

有一个很著名的论断“这句话是错的”，也展示了一种类似的悖论。它看似是一个合理的论述，但事实上不管你假设它成立还是不成立，它都与自身相矛盾。或者，还可以
116 ~ 117 来看看 Bertrand Russell 提出的问题：“小镇上唯一的理发师给（且仅给）那些不给自己刮脸的人刮脸，那谁给这个理发师刮脸？”不管你假设理发师是否给自己刮脸，都会出现矛盾。

悖论产生的根源在于自我引用。当我们构造一个句子，而该句子对自身做出断言（assert）时，就有可能出现悖论，而程序 G 就是这么做的。当进行有关自我的陈述时，这很可能会是悖论式的，从而没有任何意义。

在停机问题中，前面的悖论意味着我们无法创建一个检测是否停机的程序。如果我们试图去创建这样的程序，我们就会遇到一个矛盾。逻辑规则使得我们无法做到这一点。

目前来看，没有办法避开这种矛盾。在程序停机问题中，一个常见的避免矛盾的尝试是在 G 程序开始之前添加一个检测来判断输入是否是 G 自身的程序；如果是，G 直接返回 0。这么做的问题在于，对于任何一个函数都存在太多的（事实上是无限多的）等价程序。一个简单的例子是加法函数 $A(x) = x + 1$；它可以用 $x + M - N$ 的形式来实现，其中 M 和 N 可以是任意整数，只要它们满足 $M = N + 1$。因此，即使 G 在输入为自身的时候拒
118 绝操作，仍然会在计算同样这个函数的另一个输入 G' 时触发矛盾，因为 $G(G')$ 同样是悖

论的。

还有很多其他的很有意思的不可计算问题。在每个问题情况下，我们都发现如果该问题可计算，则所提出的方案最终总会导致可以判定停机问题，从而与停机问题的不可判定性相矛盾。这些不可计算问题的例子包括：

- 忙碌海狸（busy beaver）问题。BB(x) 代表任何一个包含 x 条指令的程序会最多移动多少步然后终止。[如果我们能够计算 BB 问题，我们也就能够解决停机问题。对于一个含有 x 条指令的程序，我们只需要模拟 BB(x) 步，如果程序终止了就说明这个程序是可终止的，否则说明该程序不会终止。]
- 全体（totality）问题。程序 $A(x)$ 是否在每一个可能的输入 x 下都会终止？[如果可以判定，我们则构造一个程序 $B(x)$，它忽略输入的 x，而针对任意给定的函数 F 和输入 y 去计算 $F(y)$。那么，当且仅当 $F(y)$ 能停机时 $B(x)$ 能在所有输入下停机。]
- 等价性问题。两个程序 A 和 B 是计算相同的函数吗？换句话说，是否对于所有的输入 x 都满足 $A(x) = B(x)$？[按如下方式构造程序 $B(x)$：它运行 $A(x)$，如果 $A(x)$ 能停机，$B(x)$ 在 $A(x)$ 终止时输出 1。构造程序 $C(x)$：它忽略输入 x 而简单地输出 1。如果有一个算法能判定 B 和 C 是否等价，这个算法也就针对 $A(x)$ 回答了全体问题。]
- 代码行问题。程序中的任意某一特定行的代码是否被执行了？[把程序修改为当且仅当 $A(x)$ 终止时执行该行。]
- 对应问题。给定两种编码方式 A 和 B，是否存在一个消息序列 $x_1, x_2, x_3, \cdots, x_k$，使得 $A(x_1)A(x_2)\cdots A(x_k) = B(x_1)B(x_2)\cdots B(x_k)$？即同一个消息序列在两种编码方式下产生了相同的编码结果。例如，令 $A = (a, ab, bba)$，$B = (baa, aa, bb)$，消息序列 3231 在两种编码方式下都生成 $bbaabbbaa$。[我们可以设计这样一种编码，它代表一个程序的单条指令执行，其中 $A(i)$ 代表第 i 条指令执行之前的机器状态，$B(i)$ 代表第 i 条指令刚执行完的机器状态。那么，只有程序终止时 A 和 B 的序列才能匹配。换句话说这个系统足够强大，使得我们可以对程序每一步的执行过程进行编码。如果对应问题存在解决办法，则说明被编码的程序会终止。]

关于程序，还有很多我们想了解的实际问题，它们也都是不可判定的。我们之所以知道它们是不可计算的，是因为一旦我们能够判定这些问题中的任何一个，我们也就可以用同样的方法去判定停机问题。

任何一个基于检验的方法，如果它声称能回答几乎任何一个有关代码的有趣问题，

它都会能够判定停机问题。阿兰·图灵基于此种观察得出结论，数学家证明定理的行为是
119 内在可计算的，因而甚至“基于检查”的方法也是可计算的。而一个试图回答关于另一个计算方法的问题的计算方法，它在回答问题时往往会陷入到对自身提出同一个问题的困境，这常常会导致矛盾。

总结

本章讲述了很多重要的基本问题，我们来总结一下其中的要点。

所有的计算方法都需要花费时间并且消耗能量，我们一般以所执行的指令数来衡量花费的时间（或能量）。

我们用数量级符号 $O(f(n))$ 来表示当 n 比较大时，一个计算所执行的指令数的增长与 $f(n)$ 呈现比例关系。这个记号忽略常数而只关注长期的增长趋势。例如一个耗时 $An + B$ 的计算其复杂度为 $O(n)$，不论 A 和 B 是多少；当 n 比较大时其复杂度增长与 n 呈线性关系。

所有的问题（函数）都可以按照它们的已知最佳算法的复杂度来进行排序。例如，排序问题是 $O(n\log n)$ 级的，因为最好的排序算法花费的时间正比于 $n\log n$。当然也有更糟糕的排序算法，例如 $O(n^2)$ 级复杂度的。

很多问题都被表述为判定问题，结果用是（1）或否（0）表示，例如程序 $A(x)$ 是否会终止？有没有访问所有城市的长度小于 k 的路径？还有一些问题表述为优化问题，例如找到最佳的背包方案或最短的旅行路线。为了一致，我们把优化问题转换为对应的判定问题。一个问题的优化表述形式，其复杂度至少不低于它对应的判定表述形式。

我们把所有能够在多项式时间内进行判定的问题划分到 P 类，把所有可以在多项式时间内进行验证的问题划分到 NP 类。对于一个 NP 问题 A，即使它唯一已知的解决方案具有指数级复杂度，但它的验证器是多项式级的。验证器可以用来寻找问题 A 的解，但真要这么做的话，需要搜索所有可能的值，所以实际上可能会比直接求解 A 还慢。关于是否 P = NP 的问题，被大家认为是数学和计算科学领域最重要的开放问题之一。

归约就是把适用于方法 A 的一个问题转化为适用于方法 B 的另一个问题的转换过程，
120 从而使得 B 的解决方案也是 A 的解决方案。“A 可归约到 B”意味着 B 足够强大，至少可以在所有输入情况下回答问题 A。

如果所有的 NP 问题都可以归约到某一个问题，那么这个问题就是一个 NP 完全问题。电路可满足性问题 CSAT 就是一个 NP 完全问题。每个 NP 完全问题都可以归约到其

他的 NP 完全问题，其中任意一个问题的具有多项式时间复杂度的算法也都可以转化成其它 NP 完全问题的多项式时间复杂度解法。目前在科学、工程、经济和社会科学等领域已知的 NP 完全问题超过 3000 个。这意味着对这些问题而言，我们已知的最好算法也至少是指数级复杂度，甚至更糟。从经验证据来看，人们似乎有理由相信 $P \neq NP$，因为成千上万的人在试图寻找这些问题的快速算法但是都失败了。

还有一大类问题是属于无法判定的，例如停机问题。这些问题的不可判定性，从根本上来讲是因为它们允许一个响应关于任何程序查询的程序提出一个针对其自身的查询，从而导致悖论。回答这些问题从逻辑上来说是不可能的。

我们面临着一些困境。我们希望计算机为我们解决的很多问题是如此复杂，以至于计算机无法在一个合理时间内返回给我们结果。还有一些问题从逻辑上就是无法计算的。因此我们很大程度上依赖于启发式（近似方法），这类方法很快，但是并不能在每个输入条件下都准确地判定或找到最优解。我们通常依赖实验性的方法，来刻画启发式方法什么时候有效、什么时候不有效。计算复杂度带来的限制迫使我们从实验性的角度去理解计算。 121

第 7 章
Great Principles of Computing

存　储

在互联网中进行搜索要比大海捞针更困难，它是在一堆针里面寻找一枚针。

—— Hubert Dreyfus

对那些按随机方式访问、但每隔五分钟就会被再次用到的磁盘页，将它们缓存起来。

—— Jim Gray & Franco Putzolu

我能找到它吗？这是计算中最常见的一个问题。我们所要寻找的信息，存放在存储器的某个地方。而存储器这么一个简单的词，掩盖了存储系统的复杂和浩瀚。如何在存储器中快速地找到我们想要的信息，以及如何将信息快速地传递到需要它的地方，这两者都非常重要。

存储系统的结构会在很大程度上影响其性能。我们以天气预报为例，来看看问题的难点。天气预报算法的设计者，将需要预报的大气层区域分成了 10 000 个小立方体，并为每一个立方体分配了位于超级计算机上的一个单独的 1GHz 处理器来进行处理。他们想知道，这个系统能否达到这组处理器联合起来对应的 10THz 处理总速度。如果能的话，他们预计可以在 3 小时内完成明天的天气预报。但是，假定在每个处理器上运行的这个算法，需要在每执行完 100 万条指令（耗时 1 毫秒）之后暂停，去从硬盘获取一组原始的天气数据，并为此花费 10 毫秒的磁盘访问时间。那么完成 100 万条指令，花费的总时间就是处理时间加上磁盘访问时间，即 11 毫秒。因此，这个系统实际的运行速度比预想的速度要慢 11 倍，将需要 33 个小时才能计算出明天的天气。这种情况下，超级计算机并没发挥太大的用处，因为它的存储系统不能足够快地把数据传递到处理器。

早在通用处理器出现之前，存储就成为信息系统中需要慎重考虑的一个问题。例如，在 20 世纪 50 年代成为一家计算机公司之前，IBM 早已是一家商业的机器公司了。他们
123 制造了一些很复杂的机器，用来在穿孔卡上选择和记录数据。一些公司使用 IBM 的机器来记录工资单、跟踪消费者信息，以及管理存货。IBM 在 1956 年引入磁盘存储，这给商业处理带来了变革的希望。因为有了这种技术，用来摆放文件柜的仓库可以融为一张小小的磁盘，还可以让中央处理器运行一些复杂的算法，来自动检索和分析数据。

计算机工程师一直都很清楚，存储方面的制约是影响计算机实际运行速度的一个重要瓶颈。程序员一直都在努力编写尽可能少占用存储空间的软件，工程师则一直在努力建造更大的存储器；与此同时，性能分析员则致力于寻找在存储空间受限条件下对计算速度进行预测的方法。

透过计算机发展的历史我们可以看到，人们一直在努力构建更高效的存储系统，使它能够跟上处理器的速度，并降低从本地的或远程的磁盘存储设备中获取数据的开销。工程师发现了用来组织存储器的三类指导原则，使得高速处理成为可能。*命名*（naming）原则涉及创建二进制串或字母数字符串，用来识别数字对象（digital object）。*映射*（mapping）原则涉及将一个名字（name）转换成处理站点与包含数字对象的存储位置之间的一个连接。*定位*（positioning）原则涉及谨慎地将数据从远程存储位置移动到临近的本地存储位置，从而实现性能的优化。计算学科中最早的科学原理之一——*局部性原理*（locality theory）——被用于优化存储系统中的信息移动。

存储系统

在关于信息的第 3 章中我们曾提到，信息可以用任何一种可保持状态并能被传感器观测到的物理量来表示。有很多技术可以适用于存储系统。表 7.1 列出了存储系统的一些基本特性。我们将在本章讨论这些特性，并指出那些能够使得原本会非常复杂的存储系统变得有条理的一些原则。

存储系统常被看作是一种仓库。我们在那里存放数据，并在之后需要的时候重新取出（回想起）这些数据。台式机上的随机访问存储器（RAM）是一个存放字节的仓库，这些字节包括能够控制中央处理器（CPU）运行的程序代码，以及这些程序所要操作的数据。硬盘或云服务器则是存放文件的仓库。但是，并非所有的存储器看上去都像仓库。对于脑神经网络模型来说，它存储的是当前传感数据和之前记录的传感数据的组合模式；在检索时，存储器返回一个与先验模式共享特性的模式，但可能与之前所有的模式都不同（详见 Kanerva 2003）。

124
~
125

存储器的层级结构是计算系统中永恒的一种特性。最顶层是访问速度最快的设备，因为只有它们能跟上 CPU 的速度。较慢的设备放在较低的层，被用来做永久存储和备份。当 CPU 需要的时候，数据将沿着这个层级结构向上移动，而当它不再被需要时，数据会向下移动。然而，顶层存储的高速度是以易变性为代价的——当设备断电时，快速存储器中的数据将会丢失。较慢的存储设备（例如硬盘）则能够一直保存它们的数据，直到这些

数据被明确擦除。层次结构带来的一种影响是，计算机性能的优化并不仅仅在于寻找使用最少 CPU 指令的算法那么简单；很大程度上，它还取决于数据在层级存储中的存放方式。

这些权衡还反映在存储设备的价格上。2014 年，访问时间为约 15 纳秒的 RAM，售价是每 GB 大约 5 美元；而访问时间为约 5 毫秒的硬盘，售价是每 GB 大约 10 美分。这两种技术的访问速度相差 300 万倍以上。而在 1960 年的时候，情形则完全不同，那时候磁芯 RAM 和硬盘存储系统的成本都高达每字节 0.3 美元，容量最大也只有 5M 字节，两者之间的访问速度差异大约是 1 万倍。

容量非常大、速度非常快并且能持续保存的存储器将来可能会出现。它们可能会依赖于一些新的技术，例如电子自旋或者有机状态。即便如此，速度与开销之间的权衡，仍然会推动存储设备的层级结构继续向前发展。

表 7.1 存储器基本特性

方面	特性	定义	例子
物理	状态	任何可以被改变和观测的物理状态	磁场、磁化块、电子自旋、磁盘上的凸点、电流方向、声音延迟线上的声音波形、阴极射线管上的荧光体、神经回路
	不稳定性	失去能量时的状态消失；非常快速的访问	随机访问存储器（RAM）、CPU 寄存器、CPU 缓存
	持续性	状态保持直到被改变或擦除；较慢的访问	硬盘、磁带、光盘
存储和获取	精确的	检索到与之前存储时完全一致的数据	RAM、硬盘、互联网云服务器
	关联的	被检索的数据与存储的数据有关联，但不一定相同	稀疏分布式存储器（Kanerva）、相连存储器（associative memory）、神经网络
	可验证性	检测之前存储的数据是否仍存在	云服务器、文件系统校验软件
访问时间	随机	访问所有存储位置需要的时间是相同的	RAM、网络包来回传输时间（大致）
	位置相关	访问时间取决于数据在媒体中的位置	自动点唱机、阴极射线管、硬盘、光盘、DVD
	序列相关	访问时间取决于在一个序列中的位置	令牌环、声音延迟线
	位置序列混合	访问时间取决于定位和随后在序列中的访问	硬盘：定位磁头，然后从磁道上读或写
访问控制	对象验证	对象管理器验证主体请求方是否有执行所申请的操作的权限	文件或数据库系统的访问控制列表，目录项中的访问控制字段
	主体验证	主体管理器为具体的对象进行操作授权	虚拟存储器映射表和能力列表中的访问控制位，对象的能力寻址

存储器的基本使用模型

我们用主体 – 对象模型来描述存储器访问的一般机制。主体（subject）是指可以对存

储的数字对象发出访问请求的任何实体。数字对象是指用来存放代表了某个事物的一组比特的容器。最为常见的一种主体是代表某个特定用户运行的计算进程（即运行中的程序）。例如，如果用户“pjd”启动了“372”进程，只有当主体“(pjd，372)”拥有对象“存储位置433”的读访问权限时，存储系统才会允许“372”进程读取“存储位置433”。

这个模型——“主体”请求对“对象”进行操作——揭示了存储系统中三方面本质：

1）对每个对象进行命名：使用一个比特或符号模式串来代表一个对象，从而将它与 126
其他对象区分开来。

2）将对象的名字映射到包含该对象的存储位置，从而建立主体和对象之间的信息流路径。

3）认证对象的拥有者是否授权允许某主体对对象进行所要求的访问。

一旦这三步完成，请求的操作就会被执行，而信息则会沿着主体－对象的路径传递。

命名

考虑到有各种各样的存储技术被实际使用，所以当我们发现存在很多种数据访问方式时，对此不应感到奇怪。存在多种访问方式的一个本质原因是，根据所处的特定环境，需要选择最恰当的数据组织方式，从而优化数据的使用。例如，程序员选择对数据进行排序，从而支持频繁的二分查找。一个公司采用标准的记录结构来存储它的员工记录，使得可以方便地检索出符合管理者所给的搜索条件的所有记录。图书馆通过杜威十进制图书分类法来组织图书和文件，使得读者能迅速地在一堆书中找到他想要的那一本。许多领域，例如医药、生物以及会计行业，已经发展出一些分类法，用来帮助识别和分类新的对象。为某种目的而设计的数据组织方式，可能在这种目的下很有效，但如果用于其他目的，则可能会导致惨痛的失败。非结构化的互联网已经令很多人大为失望，尽管它宣称可以访问全世界的信息，实际上却是一个非常糟糕的信息检索系统。

尽管处于这么多样化的环境中，命名数据的主要模式也只有六种。大多数人每天都会使用到这六种模式。表7.2给出了分类。

在前3种模式中，名字是一个长度固定的二进制串——地址（address）、地址对（address pair）或者句柄（handle）。地址通过用线性地址空间来对存储位置进行命名，常用的地址有32个比特。地址对则选取众多地址空间中的一个，然后用它来命名一个位置。句柄是文件或文件夹这类对象的全局唯一识别符，它们通常比地址要长得多，例如128个比特。句柄需要更长，因为它们必须能在整个互联网中被唯一识别。本地操作系统

生成句柄的一个常用方式是，结合本地时钟时间戳和一个特定的机器识别符，例如用于局域网的媒介访问控制（MAC）地址。以这种方式生成的句柄，一定不会短于本地时钟的长度（例如64比特）加上MAC地址的长度（48比特）。因为一台机器不会同时生成2个对象，这种方法保证了不会有其他机器生成同样的句柄。目前实际存在的对象数量远远小于句柄空间的大小——对象大约2^{20}个，而句柄空间有2^{128}个句柄——大部分可用的句柄从未被使用过。

127
~
129

表7.2 访问模式

模式	目的	映射	主要用法	例子
固定地址（n位）	在拥有2^n个位置的线性地址空间中选择一个位置；一般n=32或64	存储器硬件将地址映射到物理位置	编译器输出机器编码来表示变量和状态标签的位置；它们可以被硬件识别	RAM、虚拟存储器、机器码、IP地址
固定地址对（每个n位）	在拥有2^{2n}个位置的分段地址空间中选择一个位置；一般n=32或64	存储器被分段，每段是一个线性存储空间；地址（s，x）选择了段s的x位置	一些编译器把每个对象分配给它们各自的段；一些操作系统将所有文件视作一个分段地址空间	每个文件是一个字节线性序列的文件系统、JAVA程序中的对象
固定句柄（n位句柄）	分配给系统中每个对象的特定n位标识符；一般n=128；句柄空间是稀疏的：只有极少的句柄分配出去了	操作系统将一个句柄映射给一个对象描述子，描述子给出了对象在存储器中的详细信息	操作系统分配特定句柄给每个对象，包括对象的新版本；识别符能被管理特定类型对象的子系统识别，例如文件句柄可以被文件系统识别出来	Unix/Linux文件在创建时会给定一个唯一的“i-nodes”；所有出版者分配数字对象识别符（DOI）用于永久识别每个出版物
符号序列	用户对象用符号命名并以树的形式组织起来；一个路径名是从根节点到对象节点的一串符号名	操作系统将树表示为文件夹的层次结构；每棵树是一些各不相同的名字和它们对应的句柄列表	操作系统将路径名字映射到句柄；互联网域名系统（Domain Name System，DNS）将主机名映射到IP地址；HTTP协议将URL映射到特定主机上的文件	用户选择出现在文件夹上的名字；互联网域名是层次化的；互联网URL是路径名字
查询陈述	选择并构成数据库配对记录的公式	数据库管理系统选择所有符合查询条件的记录	匹配查询表达式的本地记录；创建列表、组合和统计摘要	商用数据库系统，例如使用接口语言MySQL的Oracle
文本串	可存于文档中的文本串	检索系统找到包含或在“语义上”接近该文本串的文档集合	找到大量集合中的相关文档	图书馆信息检索系统；互联网搜索引擎，例如Google、Bing、Yahoo

在第4种模式中，名字是一个向用户表达某种含义的符号串。这种名字的例子包括互联网主机名、文件路径名以及互联网页面的URL等。这个模式位于其他模式之上（见图7.1）。

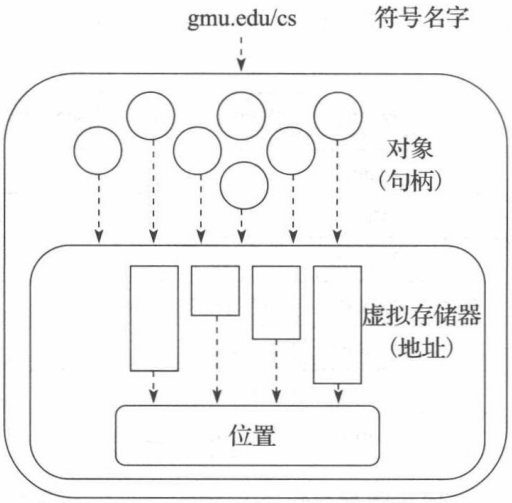


图 7.1　名字是用来代表每个个体数字对象的符号。有四种命名方式按层组织，一层叠在另一层之上，上一层的名字会映射到下一层的名字（如图中虚线箭头所示）。最上层由所有用户生成的符号名字组成，例如网页 URL。它的下一层则由创建对象的机器上的操作系统所生成的所有句柄组成。为了实现在整个互联网共享任意一个对象，句柄必须保持始终唯一。再下一层则由 CPU 执行进程时所使用的本地虚拟存储地址空间组成。最底层由物理存储器地址组成，例如 RAM 的物理页或磁盘上的记录。在本例中，符号名字“ gmu.edu/cs”指的是一个在主机“ gmu.edu”上的文件“ cs”；这个名字可能是网页读请求的一部分。互联网域名系统（Domain Name System，DNS）将主机名转换为对应的 IP 地址，而该 IP 地址上的操作系统将“ cs”转换为文件的句柄。那台机器上的文件系统将句柄映射到进程在虚拟存储器中的某个文件，以响应该网页的读请求。虚拟存储器将文件地址映射到保存了该文件的具体物理位置。动态更改映射的能力将带来重要的优势。举例来说，虚拟存储器组件（例如页）可以在不改变地址的情况下重定位。文件可以在不改变句柄的情况下重定位。数字对象的新版本可以在不改变 URL 的情况下接收新的句柄

最后的两种访问模式，将用户选择的一个符号表达式映射为一组数据对象。这个表达式用来从一个大的集合中选择一些元素，例如从企业数据库中选择雇员记录，或从图书馆中选择文档。查询（query）模式使用一种正规的逻辑语言来描述一组检索条件，用于匹配一些特定的记录而排除其他的记录。文本（text）模式则只是查询包含了给定文本串的文档。这两种模式都需要大量的搜索。例如，取决于数据库的大小，一个查询可能只花费几秒钟，也可能花费几分钟。谷歌的经验向我们表明，一旦对所有的网页内容进行预处理并生成索引，使得我们能对某个文档是否包含特定单词做出快速判断，那么我们就能在半秒之内得到一个搜索串的匹配结果。在搜索网页时，常常会有几十万个或几百万个“搜索命中”（hit）。值得注意的是，谷歌的表示算法能对命中列表进行排序，使得许多用户可以在前十个命中结果中找到有用的信息（MacCormick 2012）。但即便如此，许多用户仍然

觉得在网页中搜寻信息就像大海捞针一样难（见图 7.2）。

	Name	Salary	Sex	Seniority
1	Alice	50 000	F	10
2	Bob	45 000	M	1
3	Charlotte	60 000	F	3
4	David	42 000	M	2
5	Elizabeth	55 000	F	6
6	Fred	51 000	M	15
7	Georgia	59 000	F	12

图 7.2 数据库可以可视化地表示成一张拥有许多记录的表，每张表包含了一个标准的字段名集合。查询则是一个逻辑公式，它可以选择出匹配查询规则的那些记录。例如，“salary > 50000 & sex = F”匹配了第 3、5 和 7 条记录。“sex = M & seniority > 10”只能匹配第 6 条记录。而“sex = M & salary > 55000”则匹配不到任何记录。查询访问是通过内容或特性而不是名字来寻找信息的一种方式。一个典型的查询可以返回多个条目。数据库系统依赖于 4 个原则来避免数据的丢失和不一致：（1）数据库中的记录要复制并存放到多台服务器上；（2）事务处理是“原子的”，即对数据库进行事务操作带来的影响要在单个步骤中完成；（3）事务状态被临时记录下来，以便出错的时候可以恢复之前状态；（4）数据库以多张表的形式存储，这些表可以快速组合起来，以便响应一个查询

所有这六种访问模式都提供了*位置无关性*（location independence）：用户可以对数字对象进行操作，而无需知道这些对象具体所在的位置。自从 20 世纪 60 年代首次应用于虚拟存储系统之后，位置无关性就显现出了它的益处。在虚拟存储中，不管某个虚拟地址在层级存储结构中实际位于何处，我们通过这个虚拟地址总是能读取到正确的值。由 Sun 公司（1984）提出的网络文件系统（NFS），使得用户可以在网络的任意位置简单通过全局目录树中的路径名来访问文件。当今的云数据仓库是一种能够容忍崩溃的分布式系统，它将数据扩散到数千台其他的服务器上而不改变数据的名称。一个网页的 URL 地址，会映射到某台主机上的一个文件，而用户不用考虑主机的物理位置，以及这台主机的内部文件系统结构。互联网协议会将数据包转发给指定的服务器，而无需担心服务器到底在哪里。域名系统（DNS）将主机名转换为主机当前的 IP 地址，不管主机的地理位置在哪里。当物理位置变得不相关时，隐藏了位置的系统使用起来就会更简单、更可靠。然而，位置无关性并不总是有帮助的：移动设备上的许多应用会依赖于“地理位置”服务。

130 ~ 131

互联网搜索是六种模式中最难的一种。能够和所给的搜索关键字匹配上的对象，其数量很可能是非常庞大的——匹配的文档可能有成千甚至上百万个。URL 几乎无法提供

任何关于文档内容的线索。尽管谷歌的网页排序算法在找出有用文档方面已经做得非常出色，用户仍然经常会遇到这种情况，即排在最前面的那些文档并没有什么用。此外，有大量的信息隐藏在那些与互联网相连但存储于数据库的“深网”中，它受到密码和查询接口的保护。搜索引擎无法对它们建立索引。没有人知道这个“深网”究竟有多大，但是大多数的估计认为它占整个互联网信息的 90% 以上。因此，“整个世界的信息都在互联网上”这个想法是具有误导性的：大多数的信息并不能被搜索到，而能被搜索到的情况下还可能出现信息过载（译者注：有用信息被大量无用的相关信息淹没）。 132

映射

映射是将一个名字对应到一个位置的过程。由于存在多种访问模式和存储技术，因此映射技术也相当复杂。幸运的是，它们都以一些简单的原则为基础。

映射的基本思想是实现一个动态函数 F，使得 $F(x)$ 表示名字为 x 的对象的当前位置。映射 F 以表格的形式来存储。当一个程序请求读（或写）对象 x 时，操作系统实际上会对位置 $F(x)$ 进行读（或写）。当操作系统移动对象 x 时，它会更新这张表来记录下新的位置。将名字和位置分开，从而实现位置无关性——无论程序所访问的对象在存储器中使用了何种存储配置，这个程序都可以正常执行。

虚拟存储

自从映射原则在 20 世纪 60 年代初被提出以来（Kilburn et al. 1962），它就成为虚拟存储系统的基石。虚拟存储器被设计用来自动处理二级存储器（磁盘）和一级存储器（RAM）之间的数据移动。将这些数据的移动自动化，可以显著减少程序员的负担，从而使得他们的生产力提高两到三倍（Sayre 1969）。虚拟存储系统需要解决两个问题：

1）使每个程序拥有独立的地址空间，并使这个地址空间成为位置无关的。

2）减少在磁盘和 RAM 之间数据移动的次数。

位置无关性极其重要，因为随着数据在存储器中经常发生移动，每个具体数据项的确切位置是无法提前预测的，因此不能让它干扰程序的执行。可以用一个简化的映射原则（基于地址空间的分页）来解决这个问题。程序的地址空间被分成若干个大小相同的块，称为逻辑页（page），RAM 的地址空间也被分成相同固定大小的块，称为物理页（frame）。任何一个逻辑页都可以被载入到任意一个物理页中。进行 RAM 访问的具体过程如下：（1）程序所访问的线性地址被分解成逻辑页编号和页内地址；（2）映射表将逻辑页编号转

换成物理页编号；（3）物理页编号和页内地址重新组合，生成一个 RAM 中的线性地址偏移量。我们已经知道如何高效地完成这个过程[1]。

在多任务编程时，虚拟存储提供了一种强大的方式来划分 RAM 中的存储空间，从而
133 分配给多个程序使用（Denning 1970）（见图 7.3）。执行某个程序的 CPU 只能访问它自己的逻辑页列表中的那些页；属于其他地址空间的逻辑页是不可见的，CPU 也绝不可能生成那些逻辑页的 RAM 物理页地址。这种近乎完美的地址空间隔离是一种非常有力的信息保护机制。即便拥有足够大的 RAM 可以将所有地址空间完整地装载进内存，许多系统仍然采用虚拟存储，来达到信息保护的目的。

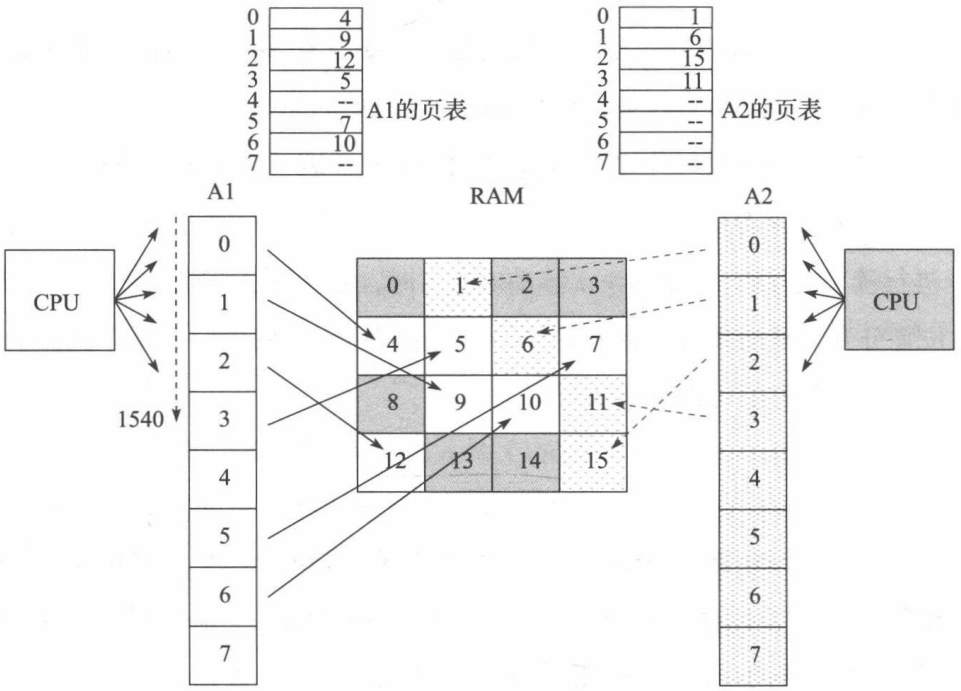


图 7.3 在本例中，两个 CPU 要在它们各自的地址空间中分别访问对象 A1 和 A2。地址空间被分成了 8 个大小为 512 字节的逻辑页，RAM 被分成 16 个大小为 512 字节的物理页（用于存储逻辑页）。CPU 使用 12 个比特来存储地址，3 比特用于表示逻辑页编号，9 比特用于表示页内地址（$512 = 2^9$）。RAM 使用 13 个比特来表示存储地址，4 比特用于物理页编号，9 比特用于页内地址。箭头表示了哪个逻辑页需要被载入 RAM。例如，左边的 CPU 的 3 号逻辑页被载入了 5 号物理页，而右边的 CPU 的 3 号逻辑页被载入了 11 号物理页。这些箭头所代表的关联被存储在每个 CPU 的页表中。在 RAM 中，页表被存储在一块不处于任何地址空间的私有区域中。左边的 CPU 要访问线性地址 1540，它位于 3 号页，页内地址为 4。根据页表，存储器映射单元（Memory Mapping Unit，未画出）将该地址转换为 2 号物理页，业内地址 4，即 RAM 中的线性地址 1028 处。RAM 里有 6 个物理页没有被分配任何地址空间（灰色的）。如果 CPU 需要访问不在 RAM 中的某个逻辑页，“页错误中断”将请求操作系统载入该页，并更新页表

最小化数据的移动是一个更需要技巧的问题。程序的整个地址空间对应有一个完整的副本，它存放在磁盘的某个文件中。在任何时候，程序的全部逻辑页中都只有一个子集被载入到 RAM 中。页表为每一页标记了它是否在 RAM 中。例如，在图 7.3 中，不在 RAM 中的页被表示为空白。如果存储映射单元遇到一个不在 RAM 中的页，它显然无法完成地址转换。这时，它将触发一个缺页（page fault）中断，从而启动操作系统的缺页处理程序来完成下列动作：（1）将缺失的逻辑页的副本从磁盘拷贝到 RAM 中的一个空白物理页中，（2）更新页表，来表明该页已经在某物理页中，（3）将控制权交还给刚才被中断的程序，它重新进行地址转换。

操作系统通过一个替换策略来决定哪些页应该留在 RAM 中。当系统需要一块空的物理页时，替换策略会选择 RAM 中不会很快被用到的页，将它与磁盘上的副本同步，然后标记它为未出现，并将它所在的物理页加到空闲物理页列表中。系统的性能与替换策略密切相关，因为每个被替换出去的页都会导致将来被访问时出现缺页中断。巧妙利用局部性原理（本章后续会讨论）的替换策略，能够最小化替换的次数，并保证最快的完成时间。精通局部性原理的程序员，可以合理安排数据访问以提高局部性，从而在虚拟存储系统中得益更多（Sayre 1969）。

共享

不幸的是，虚拟存储的这种隐藏相互地址空间的能力，妨碍了存储系统的另一个常见的目标：允许用户共享数据。一些操作系统尝试在虚拟存储上增加共享功能，但很少取得成功。

一种方法是，通过一个*操作系统接口*来提供地址空间之外的共享存储块。一组进程可以按以下方式共享存储。这组进程的主导进程向操作系统请求分配一块共享存储区，然后它将指向这个共享块的指针的副本分发给组内的其他进程。通过进一步调用操作系统，组内的成员进程可以读写这个共享的存储块。当这组进程结束时，主进程通知操作系统释放这个共享的块。这种设计存在许多问题，它很笨拙而且容易出错：很难在共享的存储区实现同步以防止竞争；如果主进程崩溃的话，就没人来释放共享存储区了；这种共享存储区只能供同一台机器上共用同一个操作系统的进程所使用。

134
~
135

上述设计有一个变种，避免了使用操作系统接口。它将共享数据载入到物理页中，在每个进程的地址空间中选择一个未使用过的逻辑页，最后将每个进程选择的逻辑页都映射到那个物理页中。这使得共享的物理页变成了这组进程的地址空间中的一员。尽管这个

方案在读写共享页时不需要调用操作系统，但带来另一个很大的缺点。如果这个共享物理页的位置有任何变化，组内所有进程的页表都需要更新。如果组内有许多进程，或者一些进程的计算被暂停，这个更新会花费相当长的时间。

一个更好的方法是为所有的共享对象定义一个新的名字空间。句柄空间可以很好地完成这个任务。能力系统（capability system）体现了这个原则。

能力

能力寻址（capability addressing）是在全局范围内共享对象的一种通用方法。能力寻址是 1966 年由 Jack Dennis 和 Earl Van Horn 共同发明的，并在随后被不断完善[2]。能力（capability）是指一个包括句柄、访问码和校验和的大型比特位模式（见图 7.4）。当一个主体向操作系统请求生成一个新的对象时，操作系统会生成一个新的能力，并将创建者作为拥有者（owner）。拥有者可以交出能力的副本，降低自身的权限，甚至规定某个副本不能被再次复制。

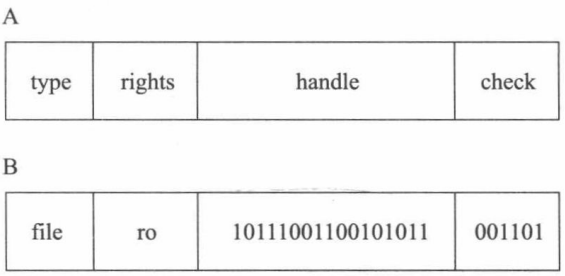


图 7.4 能力 A 是一个由 4 个字段组成的长比特模式。type 描述了这个能力指向的对象类别，例如文件、数据库记录或互联网连接。rights 字段则是一个比特串，用来说明该能力可以在给定类型上进行哪些操作。例如，对文件来说，rights 可能是 4 个比特，对应了文件操作的打开、关闭、读和写。handle 域包含了一个句柄。check 是一个加密校验和，用于确认该能力的内容自创建后没有被更改过。B 是一个文件能力的例子，表示它拥有对句柄指向的文件进行读和打开（“ro”）操作的权限。文件系统（文件的对象管理器）利用如图 7.5 所示的一张哈希表将句柄映射到文件位置。文件系统只接收文件能力而拒绝所有其他能力

每个进程都有它所拥有的对象句柄的一个能力列表。基于哈希的映射结构高效地表示了能力和对象之间的关联（Fabry 1974）（如图 7.5）。能力列表枚举了一个 CPU 能访问的所有对象，而不仅仅是内存页。而关于某个对象是否由内存页来实现，诸如此类的细节则被交给处理该类型对象的子系统来解决。

能力寻址中的一个关键假设是，能力的持有者拥有能力中所指定的那些权限。对象管理器只是简单地接收能力，它不检查访问控制列表来判定调用者是否有权限。理论上

讲，一个侵入系统内核的黑客有可能更改能力或者进行非法复制。但是发生这种损害的可能性足够低，使得这种系统是可用的。 136

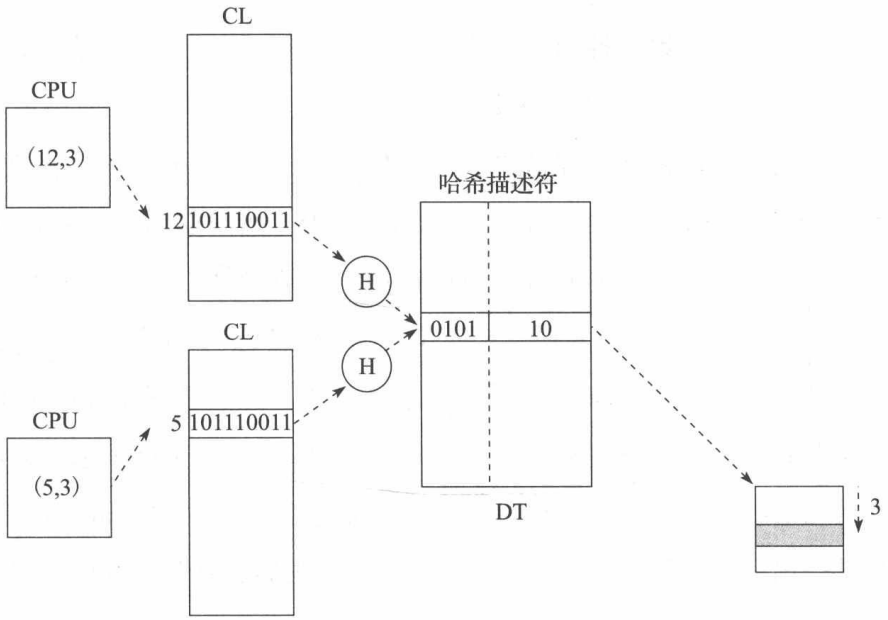


图 7.5　两层映射结构使得我们可以共享数字对象。描述符列表（Descriptor Table，DT）是唯一列出每个对象的位置的表格。数字对象唯一的句柄存储在能力中，而全部的能力被存放在一个能力列表（Capability List）中。哈希函数 H 将句柄映射到一个较短的哈希码，这个哈希码可以定位对象的描述符，而该描述符指向了目标对象的存储地址。在这个例子中，顶部进程将共享的对象记为能力 12，而底部 CPU 则将之记为能力 5。两个 CPU 都请求对象的第 3 行。哈希函数将句柄转换成一个 4 位的哈希值并作为 DT 的索引。DT 中的项包含了一个指向位置 10 的描述符，也就是对象的基地址。如果操作系统因为各种理由重定位了这个共享的对象，它只要简单地更新描述符来表示新的基地址就可以了，所有参与共享的进程都会被定向到新的位置

能力系统在“约束”方面非常有效，这意味着可以确保每个计算进程只有与自己的工作相关的对象的权限，而没有其他的权限。能力系统具有高容错性，因为错误会被约束在一个小的保护域内而不会通过系统进行传播。能力寻址是实现诸如 Java、Smalltalk 和 Python 这类面向对象系统的一个重要的原则（Wulf et al. 1974，Miller 2003）。因为能力可以作为参数在网络中客户端和服务器端之间的消息中传输，它可以被轻易地扩展到像 Amoeba（Tanenbaum & Mullender 1981）或 Tahoe-LAFS 这样的分布式环境中[3]。能力中的加密校验和使得它们很难被伪造，这意味着服务器可以信任它们，并将它们视作客户端拥有权限的证明。

能力寻址原则在互联网中被独立地发明并用于实现数字对象，其动机是克服标准的

网页 URL 所具有的局限性。常见情况下，对象的拥有者改变了对象的内容，将它移到了另一台主机上，甚至可能删除了这个对象。用户如果很意外地发现对象的内容被更改了，或者重要的对象突然消失，他们多半会很不高兴。在第一种情况下，共享对象的内容可能不是用户所期望的内容。在第二种和第三种情况下，没办法通知用户说某个对象有了新的 URL 或已经永久消失了。对于发布者来说，他们希望自己的工作能一直被成功地引用和访问，因此上面的局限性就是不可容忍的了。在不修改网络节点的操作系统的前提下，为了解决这些问题，Robert Kahn 和 Robert Wilensky（1995）为互联网提出了能力寻址原则。他们开发了一个名为 handle.net 的服务，可以让用户注册句柄以及对应的 URL。用户可以通过发布这些句柄来共享对象（见图 7.6）。句柄系统提供了一种方式来访问任意注册过的对象，即使过去多年后对象被转移到了新的服务器上，或是获得了新的 URL，仍然不影响大家对对象的访问。国会图书馆创建了一个数字对象标识（DOI）系统，可以让所有出版者为他们的出版物生成句柄，并在图书馆中注册它们。图书馆依赖出版者来维护它们的内容，如果某个出版者停业了，并且没有将它拥有的内容转移到别的服务器，那么这些内容会变得无法访问，因为它的 DOI 不指向任何地方（Denning & Kahn 2010）。

137
~
138

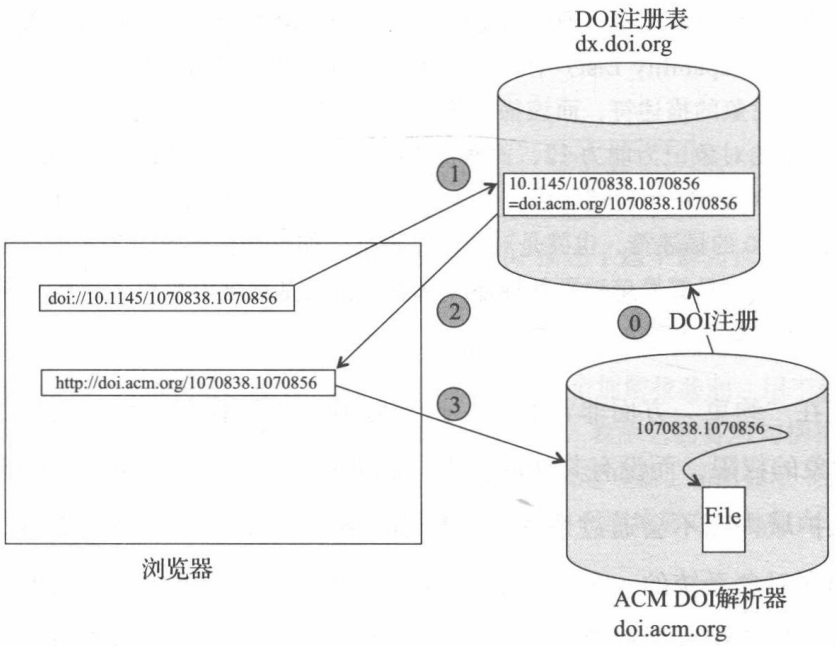


图 7.6 数字对象识别（Digital Object Identifier，DOI）系统给互联网带来了句柄系统的一些特性。以 ACM 的数字图书馆这个专业社团为例，访问协议由 4 个步骤组成。一开始，ACM 为一个新出版的对象生成一个 DOI，这个 DOI 包括了一个 ACM 的唯一数字（10.1145）和一个 ACM 选择的唯一字符串。然后，用户可以从引用中得到这个 DOI 并请求注册表解析它（第 1 步）。注册表返回 ACM 数字图书馆中该对象的 URL（第 2 步）。ACM 数字图书馆解析器定位到具体的对象（第 3 步）

认证

从很早以前，信息保护就是操作系统设计者关心的一个首要问题（Dennis & Van Horn 1966，Wikes 1968 a，Denning 1971，Lampson 1974，Saltzer & Schroeder 1975）。在将名字映射到对象的这个环节，操作系统需要确认发出请求的主体确实拥有执行该操作的权限。这个确认的过程被称为访问控制（access control）。

一般来讲，对象的拥有者（它的创建者）会声明谁能够以何种方式访问它。一个授权是一个形式为“主体 s 可以在对象 x 上执行函数 f”的陈述，可以简写为（s, f, x）。一个主体通常是一个用户和该用户所拥有的某个进程的组合。例如，s = (Ann, 317) 指 Ann 在执行的 317 进程。一个类似于“Ann 的任何进程都有读文件‘abc’的权限”的授权可以简写为 ((Ann, *), read, abc)，其中 * 表示任何一个进程。可以执行的操作取决于对象的类型。例如，读操作只能用于文件或内存页，而暂停操作只能用于进程。每个对象管理器都有责任阻止那些可能违反授权的访问。例如，如果只有 ((Ann, *), read, abc) 这个授权，文件系统将阻止 Ann 对“abc”进行写操作访问。

访问控制列表（ACL）是用来表示授权的一种常用方式。一个对象 x 的拥有者会创建 ACL[x]，而 ACL[x] 中的项具有 (s, f) 这样的形式。只有当 ACL[abc] 包含 ((Ann, *), read) 这个项的时候，文件系统才会允许 Ann 去读文件“abc”。目录系统和数据库系统会对每个目录项或数据库记录关联一个访问控制列表。

Unix 系统将 ACL 压缩成 9 个比特。它假定只有三种用户：拥有者、拥有者所定义的组中的成员、普通公共用户。它为每种类型的用户定义了三个权限比特：读（r）、写（w）、执行（x）。这样得到的 9 个比特代码存储于对象的目录项中，而不是在一个单独的 ACL 文件中。例如，Ann 声明她不能执行她的文件“abc”，她的组只能读这个文件，而其他的普通公共用户可以读和执行这个文件，这样的授权会以访问代码（rw-r--r-x）的形式存储于文件“abc”的目录项中。

139
~
140

表示授权的另一种常用方式是使用能力列表（例如页表或对象列表）中的访问码。当操作系统创建一个主体 s 时，同时会创建它的能力列表 CL[s] 并让它指向所有 s 可以访问的对象。这样，Ann 对文件“abc”的读权限会在 CL[Ann] 中存储为 (read, abc)。ACL 和 CL 经常一起使用：当文件载入地址空间时，能力列表就是根据 ACL 中的授权来进行初始化的。

一些军用或政府系统会给每个用户标记许可（clearance），给每个数字对象标记安全

标签（诸如未分类、机密或顶级机密之类）。许可与标签之间的关系定义了额外的访问限制。用户既不能读取标签级别高于他们的许可级别的对象，也不能向标签级别低于他们的许可级别的对象写入数据。这种系统根据主体所允许生成的信息流来控制对象的读写访问（Bell & LaPadula 1976，Denning 1976）。

为了让权限管理和其他操作变得更容易，多数系统都有一个拥有系统中所有对象的所有权限的“root”账户（超级用户或管理员）。只有一小部分受信任的管理员可以登录这个账户。对于大部分系统来说，root 账户的存在是一个主要的安全风险。基于能力的寻址方式的一个优势就是不需要超级用户的存在。

层级结构中的定位

定位（positioning）是指将数据放置到存储器层级结构或者网络节点的不同层上，以保证系统具有更好的性能。定位会极大地影响系统的性能。回想一下本章开头的那个例子，由于存储系统传递传感数据到 CPU 的速度不够快，我们所设想的那台超级计算机只能以 CPU 额定速度的 1/11 来运行。

代价分析是所有解决定位问题的方法都要遵循的一个原则。如果某个数据块会被多次使用到，而在 RAM 中保留这块数据的代价比从二级磁盘读取这个块的代价要小，那么我们就选择在 RAM 中保留这个块。这个情况可以用下述公式来表示。令 R 表示数据块重新使用的时间间隔，B 表示块大小，U 表示 RAM 中每个字节每分钟的单位花销，D
141 表示从磁盘读取一个块的花销。当 $UBR<D$，即当 $R<D/UB$ 时，在重用时间间隔内一直保留这个块所产生的花销会更小。使用 1985 年 Tandem 计算机系统的花销和块大小，Gray 和 Putzolu 将 R 的阈值设置为 5 分钟。换句话说，如果 5 分钟内某个数据块需要再次在 RAM 中用到，那么就在 RAM 中保留它，否则就将它保存在磁盘上去。二十年后，同样的规则被用于硬盘和 RAM 之间的大块数据，以及闪存和 RAM 之间的小块数据的管理（Graefe 2007）。

这个思路可以重新描述成最优化准则。对于一个系统来说，参数 D、U 和 B 是固定的，我们可以把它们组合成一个决策阈值 $T = D/UB$。所以在刚刚使用过一个数据块之后：

1）如果距离再次使用该块的时间超过 T，则马上从 RAM 中移除这个块。

2）否则，保留这个块直到下一次再使用它。

在下一次使用之前，我们不会将一个块保留一部分时间之后再移除出去，因为这只

会增加这段时间内 RAM 的花销，而又无法避免从磁盘中再次读取。

这个规则在每一次数据使用之后都适用。因此，如果重用某个块的时间点在阈值窗口内，同样的规则仍会被使用，即使该块已经被载入。因为这个规则会单独作用到每一个被用到的块，所以 RAM 的总使用量可能会变化。如果有很多块在 T 时间内会被重用，那么 RAM 的分配会增加，而如果被重用的块少，那么 RAM 的分配会减少。1976 年，Prieve 和 Fabry 定义了 VMIN 策略，正是用上面这个决策规则来解决变量空间最小化问题，并证明它是最优的。对于一个给定的平均 RAM 分配量，没有其他的策略可以使得块载入量更小。

这个原则在实践中应用起来并不容易，因为数据块的个数可能很大，而它们再次被使用的时间是随机的。此外，关于是否应该保留一个块直到它被再次使用，目前做出的决策是不准确的，因为我们并不能预见未来。为了针对再次使用的时间产生一些有用的预测，我们需要一个预测程序如何使用它的代码和数据的模型。

在 20 世纪 60 年代虚拟存储被提出之后，针对存储使用的预测模型研究也开始了。虚拟存储非常依赖于它的页替换策略——这个策略决定了在 CPU 遇到缺页事件时，RAM 中哪个的物理页将被替换出去。不同的替换策略反映的是不同的预测模型。虚拟存储的性能与预测模型的选择息息相关。

1966 年，IBM 的 Les Belady 开展了一项具有深远影响力的研究，分析了固定尺寸 RAM 上的替换策略的性能。他得出结论，最近最少使用（Least Recently Used，LRU）策 142
略总是比其他策略的性能好。而这个 LRU 策略每次都选择将最久没有用过的那个页替换掉。作为参照，他定义了 MIN 策略即最少缺页错误准则，该准则每次选择将来再次使用的时间最远的那个页。MIN 能得到比其他固定划分策略都要少的页错误（Aho et al. 1971）。不幸的是，MIN 无法真正被实现，我们无法准确知道未来的内存访问行为，对此也没有一个非常好的预测模型。LRU 策略和 MIN 策略之间还有相当大的性能差距，寻找更接近 MIN 策略的研究仍在继续。

研究人员使用引用映射表（reference map）来帮助他们记录程序在运行时究竟如何使用存储器。引用映射表是一个按照时间顺序排列的地址空间样本序列，表示了每个采样时段内哪些页被用到。这些映射表揭示出一些惊人的模式，即对存储器的引用经常会集中在一个小区域内，并持续较长的时间。对于每个程序来说，这些模式是各不相同的，就像声纹一样（见图 7.7）。引用映射表中所使用的区域，代表了最大化系统吞吐量的时间 – 空间印迹（见图 7.8）。

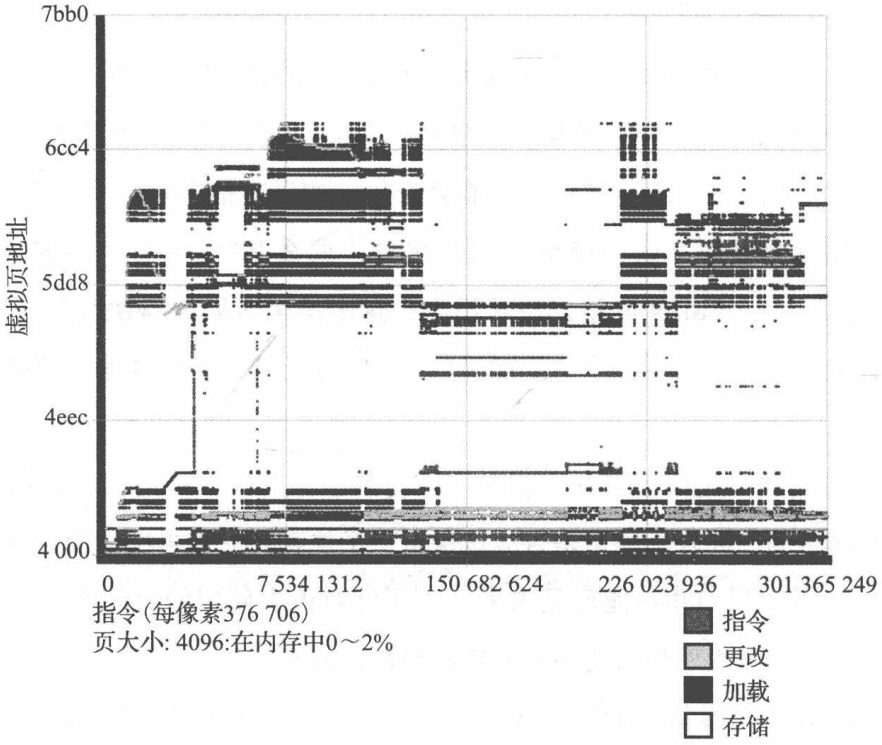


图 7.7 这是一个 Linux 系统下 Firefox 网页浏览器的页引用映射表。横轴表示虚拟时间，通过存储器的引用量来衡量（每个像素大约是 380K 次引用），纵轴代表虚拟页地址。较暗的像素表示这个页在长度为 380K 次引用的窗口中被引用了。映射表揭示了程序的位置集合，并直观地表明这些位置集合在较长时间内是稳定的，尽管也会在某些时间点从当前位置集合转到其他位置集合。在这幅图中，一个样本区间中的位置集合通常持有 30 至 60 个样本。在超过 97% 的情况下，当前样本区间中包含的页可以近乎完美地预测下一个样本区间中包含的页。这些引人注目的图说明了每个程序都有它们特定的局部性行为。程序使用代码和数据的方式并不是随机而无规律可循的

程序对存储器的访问常常长时间地集中在地址空间的一些子集中，这个倾向被称为*局部性原理*[5]。如图 7.7 所示，我们可以将某个计算程序的存储访问请求描述成一个序列

$$(L_1, P_1), (L_2, P_2), \cdots, (L_k, P_k), \cdots$$

其中每个 L_k 代表一个位置集合，即地址空间中对象的一个子集，每个 P_k 代表一个时段，即位置集合持续的时间。如果已知位置集合和时段，我们就可以让每个位置集合在它的时段内保留在 RAM 上，从而解决定位问题。很快我们就能看到，这个做法是非常接近最优解的。

工作集（working set）是一种用来跟踪程序的局部存储访问行为模式如何变化的测量工具（Denning 1968a）。工作集标识出，在刚过去的一个长度为 T 的虚拟时间窗内，

地址空间中的哪些页被使用过[6]。我们希望使用一个较小的时间窗，它只要刚好长到可以对目前的位置集合中所有的存储页进行采样。许多研究（例如图 7.7 所示）都表明，一个合适的采样窗口通常只占时段长度的一小部分。有了这样一个窗口，工作集就变成了一个极佳的预测器，可用来预测接下来的位置集合，并使系统性能接近最优（见图 7.9）。

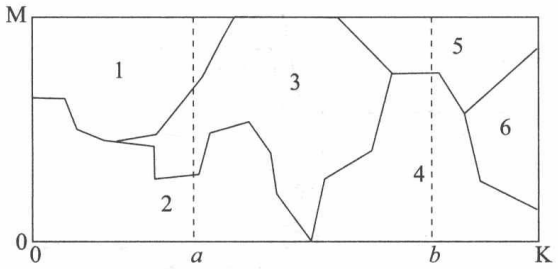


图 7.8　性能分析员从时间 – 空间角度评测存储器的使用情况，即查看程序随着时间的变化使用存储器的页数 – 秒数曲线。这里我们可以看到一个大小为 M 的存储器在 K 秒内的变化。6 个区域表示了 6 个程序使用存储器的时空情况。在时间 a，程序 1、2 和 3 在存储器中；在时间 b，程序 4 和 5 在存储器中。存储器的时空开销与系统的吞吐量有关，即那个重要的定理 $M = XY$，其中 X 是每秒内程序完成的吞吐量，Y 是程序的平均时空开销（Buzen 1976）。在这里吞吐量是 $X = 6 / T$，因为有 6 个程序被执行完，平均时空开销就是总的可用时空开销 MK 的六分之一。理想情况下，如果每个程序的引用映射表足够小的话，程序数量的最大值就会占据整个可用的大小为 MK 的时空，从而最大化吞吐量

当我们把局部性原理应用于多任务编程时，这种原理告诉我们对每个程序来说最理想的 RAM 分配就是它当前的工作集。只有当一个程序的工作集可以被全部装载到 RAM 中空闲的区域时，程序才应该启动。遵循这个原则的虚拟存储系统不会发生抖动（thrash）[7]。

当把局部性原理应用于 CPU 缓存时，它告诉我们要根据 LRU 原则来替换缓存项。这是因为最近最少使用的页很可能是属于过去的局部位置集，它在后续运行中不再那么相关。这样的话，缓存中就保存了那些最可能在接下来会被使用到的页，从而模拟了 MIN 策略。

当我们把局部性原理应用于互联网时，它告诉我们要将网页的副本存放到靠近用户簇的“边缘缓存”（edge cache）上，例如一个局域网中。一个用户簇会有它自己的局部行为，而边缘缓存将局部页定位到靠近它们被处理的地方。在互联网中，一些热门网站可能会积压一个很长的请求队列，这会阻碍网站的服务从而导致长时间的延迟；而缓存可以打破这种集中式服务器的长队列。Akamai Technologies 尤其擅长解决这方面问题：他们的

143 ~ 146 边缘缓存为30%的互联网传输流量提供服务。他们的算法会评估请求从哪里来，并计算出放置缓存的最佳位置。就像在个人计算机中一样，互联网中的局部性是可以预测的，而局部性模型可以显著地提升性能。

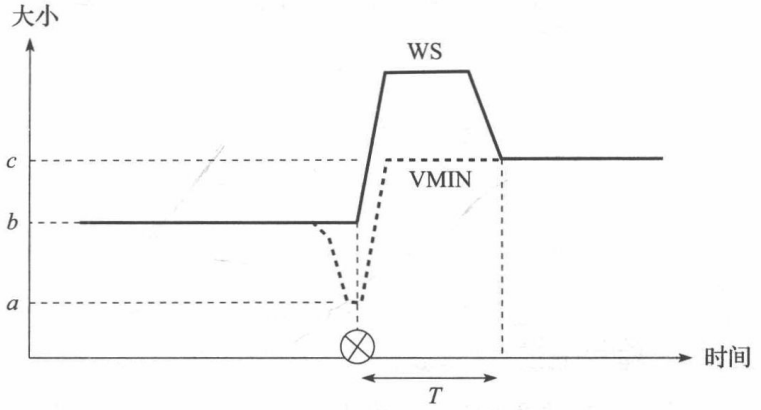


图7.9 当存储器分配随着位置集合变化时，Belady的MIN并不是最优的，VMIN才是（Prieve & Fabry 1976）。对于一个给定的T，VMIN直接查看此页的下一次引用；如果这个页会在时间T内再次用到，那么VMIN就在整个区间内在RAM中保留它直到它被再次用到；否则VMIN就马上删除它，等到页错误发生之后再载入。参数T的选取需要折衷考虑保留一个页的开销和再次载入它的花销。当T小于片段长度时，VMIN和WS（Working Set）都可以看到并保留位置集合。它们之间的唯一区别发生在片段转移时。本图给出了从一个小的位置集合（大小为b）到一个大的（大小为c）且有部分重叠（大小为a）位置集合的转移。当这个转移开始的时候（带圈的“x”），WS在旧的页的基础上累积了新的位置页；在T时间后进入新的片段，WS就只能看到新的位置页了。VMIN通过在上一次使用后就马上移除页来预测转移。在转移发生之后的T个单位时间内，WS和VMIN之间相差了$b-a$页，直到时间窗口T结束，WS衰减到和VMIN一样。因此，如果旧的片段持续P，那么时空的相对差值最多是$(b-a)T/bP$，小于T/P。因此，如图7.7所示，如果T是片段长度的3%，WS距离最优方解的偏差不超过3%

为什么局部性是基础

许多实践经验表明，局部性是所有计算的一个基础特性。通过引用映射表可以观察到大量的程序行为都表现出位置-时间的局部模式，并证实了使用较小时间窗内的工作集，能准确地观察到局部位置集合。此外，CPU缓存的全面成功以及互联网中的边缘缓存，也是对局部性的强有力验证。

局部性行为是否会是由于编译器优化而导致的一种人为结果呢？例如，是否有可能是由于编译器的策略，使得相互引用的代码块和数据聚集在了一起，从而造成了这样的行为？ Madison和Batson（1976）证明了，局部性行为在程序的源代码中就已经

存在了。因此，局部性的行为模式看来是由人类解决问题的方式本身带来的。例如，很常见的分治策略就确保了算法会将引用聚成更小的代码和数据子集，以解决下一个阶段的问题。使用线性数组、字符串或者向量的算法，通过循环来访问这些对象，从而形成了局部性的行为。优秀的编译优化算法则会保留局部性，使得存储系统能高效地进行操作；相反地，糟糕的优化算法可能破坏局部性，使得存储系统的使用变得很低效。

局部性原理事实上可能具有比上述讨论还要更深远的影响。2010 年，Yuri Gurevich 发表了一篇报告来试图回答“算法是什么”。他将算法公式化地定义为对一个可以在数据上执行操作的智能体的控制。在对于该智能体所允许的操作的要求中，一个要求是边界原则：一个操作只能改变数据结构中的一个有限的、带边界的区域。换句话说，一个计算方法必须要服从局部性原理，才能被看作一个算法。

结论

与处理一样，存储是计算的基础。在存储系统中，代码和数据都是层级化存储的，高速设备的容量相对较小，而低速设备的容量相对较大。高速设备允许实现高速计算，低速设备允许存储大型数据集并永久存储。由这些存储系统构成的庞大网络在互联网中是相互连通的。 147

存储系统的可用性取决于四类问题的解决办法：命名、映射、认证和定位。*命名*是指分配字母数字或比特位串来标识对象。*映射*是指将一个名字和一个保存了该对象的存储位置关联起来。*认证*是指确认发出请求的主体是否具有在对象上执行所请求的操作的权限。*定位*是指将数据存放到层级存储结构或互联网中的某个位置，从而优化性能。

存储一共有六种主要的访问模式——地址、地址对、句柄、路径名、查询和文本搜索。每一种都有特定的目的，这些模式在大多数计算系统或互联网上都存在。实现这些模式的系统可能相当复杂。

定位是基于下面这样一种原理：如果在再次使用之前保留一个数据对象的开销要低于之后从另一个位置重新读取它的开销，那就应该将它保留在原处。在许多系统中，最典型的例如虚拟存储，定位决策是由这些开销的比例来决定的。局部性原理——计算时的引用会聚集在地址空间中的较小子集中——发展成为一个被完善证明的理论，用于预测哪个对象最有可能在接下来会被使用。预测所给出的位置，应当被放置到离操作点近的地方。局部性理论促进了存储系统的设计和性能优化。

近年来有人作出预测，虚拟地址技术终将消失，因为随着 RAM 技术的不断进步，大部分的程序将不再需要分页。一个没有分页而更简单的操作系统，将给所有人提供他们所需要的所有 RAM，但这种愿望恐怕不太可能会真的发生。虚拟地址技术仍然会存在，因为它们解决了共享、命名、认证和防止程序相互干扰等一系列重要问题。即便它们的数据定位策略不被使用，它们也仍是不可或缺的。此外，我们想要运行的很多算法都涉及处理“大数据”——这意味着数据的规模超出了我们目前的存储系统的容量——RAM 永远都是
148 不够用的。

第8章
Great Principles of Computing

并　行

我们把整个系统组织成一个由若干顺序过程组成的社会，它们之间的和谐协作由显式的相互同步声明来控制。

——Edsger W. Dijkstra

商业化多核芯片的普遍使用正迫使计算思维并行化。

——Walter Tichy

合众为一。

——美国国家格言

当我们想起计算时，通常想到的是控制单个 CPU 执行一个指令序列的单一进程。我们对于算法的很多定义都强调一步一步地进行——每次只做一件事情。

但是在现实生活中，我们会同时做很多事，并与很多人交互，大家都在同时做很多事。在日常行为中，在移动和桌面的操作系统中，我们都能看到这样的情况。我们在互联网上蜂拥发起的信息处理过程中可能会有一些顺序执行的成分，但其中大量存在的还是多个个体并发运行的情形。顺序的过程已经无法充分地描述现代的计算世界。这是一个由很多独立自主的个体组成的纷乱世界，每一个个体都试图既达成个体目标，又实现共同目标。

我们该如何描述和管理这样的计算？

我们将在“并行计算”（parallel computation）这一标题下进行讨论，这个术语通常指计算是由多个并发个体协同完成的。该术语隐含地表明了“串行计算”(serial computation)是它的一个特例，可以作为并行系统的基本构成单元。然而，还是有很多计算，即便在很细的粒度，也很难辨别出其中的顺序成分。

并行计算的设计师主要处理两大类现象：

1）协作并行：多个进程（自主计算个体）相互同步完成一个共同的目标。例如，拥有 10 000 个处理器的超级计算机开始一项由 10 000 个进程组成的天气预报计算，每个进程运行在一个处理器上，能够以比单个处理器快 10 000 倍的速度完成整个计算。

2）竞争并行：多个进程，它们之间很少或没有相互同步关系，同时利用一个网络的

有限资源来实现各自的目标。多个进程请求同一资源时将按序排成队列，具有长队列的资源就成为限制单个进程响应速度的瓶颈。例如，在紧急情况下，手机网络可能不堪重负，用户将经历长时间的等待才能拨出电话。

这两种类别并不是相互独立的。例如，一些超级计算机操作系统允许多个进程竞争少于进程数的处理器。通过将每个查询划分为数千个子查询分发到数千个处理器上，谷歌避免了其服务器上每天上十亿查询负载所产生的瓶颈，这些处理器协同工作使得每个查询在 0.5 秒的时间内就可得到响应。

本章我们讨论协作并行，而在下一章排队（queueing）中，我们将讨论竞争并行。

并行计算的早期方向

20 世纪 40 年代，计算机工程师主要关注带串行处理器的计算机，这种处理器按顺序每次执行一条指令。他们认为这是实现可靠计算机的最佳路径。那个年代，对于这样一个新兴技术来说，并行计算机和并行算法实在是太过复杂了。

即便是这样，很多计算机工程师仍然一直在思考并行的可能性。实现 CPU 的电路中包含了许多传输信号的并行通路。工程师知道竞态条件（race condition）是影响可靠性的主要因素。当多个输入信号在并行通路中传输，且输出的值取决于各并行通路的速度时，就出现了竞态条件。例如，设计者计划使输出值保持为常量“1”，但如果较快的信号先
150 于较慢的信号到达输出，该输出值可能会产生短暂波动变为“0”。这种波动可能导致准备接收该输出值的下游电路出现故障。Maurice Karnaugh（1953）示范了一种逻辑电路设计技术，可以避免由竞争信号导致的波动。遗憾的是，Karnaugh 的方法无法很好地扩展到超大规模电路中。

硬件工程师往往在大规模电路中内置时钟以保持其稳定性。他们把机器构造成一个拥有一组 1 位触发器作为反馈的逻辑电路（如图 8.1 所示）。在一个时钟节拍脉冲中，逻辑电路的输出被写入触发器组中。时钟节拍之间的间隔时间比通过逻辑电路最慢路径的延迟要长，这样用户就不再会观察到由逻辑电路内部的竞争引起的波动。

遗憾的是，时钟电路难以扩展。在 3GHz 时钟的节拍内，光可以传播 4 英寸[⊖]，意味
151 着信号有足够的时间遍历一个典型的 3cm 的芯片。而对一个电路板尺寸为 1 英尺[⊜]的电路来说，我们则需要把时钟频率降到 1GHz 或更小。

⊖ 1 英寸 = 0.0254 米。

⊜ 1 英尺 = 0.3048 米。

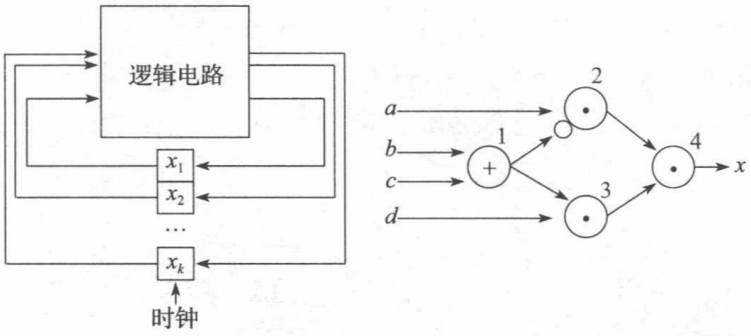


图 8.1　CPU 中的逻辑电路与一组 1 位触发器（x_1，x_2，...，x_k）相连，触发器的信号表示电路的最新状态（左图）。每来一个时钟脉冲，触发器就被置为该电路输出的当前值。触发器的新值又传输到该电路作为输入，进而产生一组新的输出。然而，当新的信号在电路中传输时，输出可能暂时改变。为了避免错误的输出被触发器读到，时钟节拍间隔被设置成比电路稳定时间长一些。右图所示的逻辑电路有一个或门（1）连接到两个与门（2、3），之后再连接到另一个与门（4）。从门 1 到门 2 的输入要经过非门，表示从门 1 出来的信号要取反。如果该电路的状态 $abcd = 1001$ 或 1011，则输出 $x = 0$。然而，当 c 从 0 变成 1 时，若通过门 2 的传播比门 3 慢，则因为两个门之间的时间差别，将导致瞬时的 $x = 1$

很多电路设计者试图通过建造模块化的电路来缓解这种信号传播问题，时钟仅应用于各个模块内，在模块之间采用异步信号传输（不采用时钟同步）。例如，CPU 盒和磁盘控制器盒内可以有各自的内部时钟，并采用就绪 - 确认（ready-acknowledge）协议来交换请求和数据。在就绪 - 确认协议中，发送方把要发送的数据放到缓冲区，然后向接收方发送一个就绪信号；当接收方收到数据之后，向发送方返回一个确认信号。这个协议可以以任意长度的周期在发送方和接收方之间工作。Ivan Sutherland（2012），异步电路最初的倡导者之一，在 2012 年指出，如果希望继续扩大电路的规模并降低能耗，那么我们比以往任何时候都更需要把时钟从电路中去掉。

很多设计者期待利用并行这种更积极的方式来加速计算，而不是依赖加快时钟。如果一个待解的问题能被分解为多个较小的、独立的任务，每个任务都能运行在一个不同的处理器上，那么 N 个处理器只需花费单个处理器所需时间的 $1/N$ 就能完成所有工作。大规模并行的超级计算机已利用这一原理推进到了一个巨大的 N 值，有些超级计算机甚至可以达到百万个处理器（见图 8.2）。

遗憾的是，很少有问题可以分解为完全独立的多个部分。我们来看这样一个例子：一个高度简化的天气预报算法，它用一个二维网格表示整个国家。计算时，每一个网格单元记录自己的气压、风向和风速。每一个网格单元与相邻的四个单元——记为东、南、西、北——相互作用，四个相邻单元的值对其产生影响，形成该单元新的气压、风向和风速

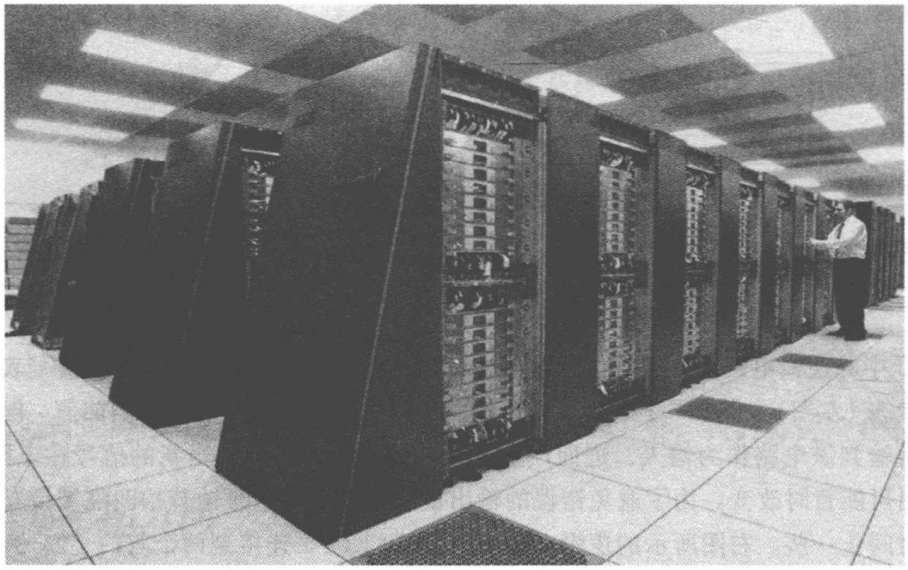


图 8.2 美国 Argonne 国家实验室的 Blue Gene/P 超级计算机拥有超过 250 000 个处理器，这些处理器通过高速光纤网络相连，部署在 72 个机架中。这是典型的现代“集群”（cluster）计算机，通过大规模并行处理达到极高的计算速度。另一种常见的超级计算机的结构是“网格”（grid），这里，互联网上成千上万的计算机被动员起来，利用其空闲时间完成大型计算任务中的各个片段。2013 年的最快计算的世界记录是由美国橡树岭国家实验室（Oak Ridge National Laboratory）的克雷泰坦超级计算机拥有，它每秒钟可完成 18×10^{15} 次浮点计算（18 petaflops，即每秒 18 千万亿次浮点计算）。Michael Flynn（1972）将这种体系结构描述为 MIMD（Multiple Instruction Multiple Data，多指令流多数据流），并预言这种结构能够达到最快的速度

值。只考虑气压计算：令 t 表示计算中的一个时间步长，$P(t)$ 表示时间步长 t 内一个单元的气压。某个单元新的气压值可以通过计算前一个时间步长内其四个相邻单元气压的平均值获得[1]：

$$P(t) = [P_{北}(t-1) + P_{东}(t-1) + P_{南}(t-1) + P_{西}(t-1)] / 4$$

时间步长之间允许的时钟时间是：网络传送四个相邻值的时间，加上该单元计算四个值之和并除以 4 以及存储结果的时间。因为所有的通信都是在本地进行，因此在同一个
152 时钟时间内，这种计算方式可以扩展到大量的单元。

我们用一个由 N 个处理器组成的机器，每个单元都有一个单独的处理器，来计算所有时间步长 $t = 1, 2, 3, \cdots$ 内的气压分布图，其计算速度可以加快 N 倍。通过时间的不断演化，我们就可以预测未来的气压值。

为单处理器计算机编制的普通程序会是什么情况呢？在具有 N 个处理器的机器中它们能否被加速 N 倍？是否有可能打造并行化编译器，将普通程序翻译为能够在多处理器

计算机运行的并行程序，而无需再培训程序员？20 世纪 60 年代，IBM 的一位计算机架构师吉恩·阿姆达尔（Gene Amdahl），想知道并行化编译器能做到什么程度，他给出了一个公式用来计算这种加速比，这就是现在我们所熟知的阿姆达尔定律（Amdahl's law）。假设一个程序中可并行部分的比例为 P，并且这部分能够达到 N 倍加速，则并行化后的 153
程序的运行时间是之前的 $(1 - P) + P / N$ 倍。例如，如果一个程序中有 10% 的部分可并行化，现有 10 个处理器可用，则该程序并行化后的运行时间是单个处理器上运行时间的 91%。阿姆达尔认为很多常见的程序由于包含过多的串行依赖，因而无法从并行处理器中得到太多的好处。

由于摩尔定律（Moore's law）持续起作用，芯片的处理速度大约每 18 个月翻一番，因而并没有人抱怨计算的速度。然而，2000 年左右，芯片制造商开始遭遇如前所述的扩大时钟控制芯片规模的困难。他们转向并行处理以保持摩尔定律所承诺的翻番速度。他们将两个并行的 CPU（称为“核”）放在同一个芯片上并保持原有的时钟频率，而不是通过缩减特征尺寸和时钟频率加倍来使得芯片的处理速度翻番。之后十年内，16 核的芯片已经可常规应用。但是这里有一个问题：开发应用软件的程序员必须学习如何为这种新的芯片编写并行程序。他们可以利用什么原理来做好此事呢？

自从 20 世纪 60 年代以来，操作系统设计师已经积累了大量的构建有效并行系统的知识。操作系统实现了应用程序的并发进程，并按照进程在“就绪列表”中的等待顺序将各并发进程分派到各 CPU 上执行。操作系统非常擅长在有限的资源集中并行运行多个用户进程。

很多关于串行算法设计和调试的直觉在并行编程中完全失效了。并行编程极可能带来竞态条件以及时序依赖的间歇性错误，同时，当进程交换信号或因为死锁造成混乱时，也会带来一系列同步问题。计算思维不得不从串行计算转向并行计算。

并行系统的模型

从 20 世纪 50 年代到 70 年代，虽然业界没有给并行程序设计太多空间，但一些研究人员仍积极开展研究，并不断丰富并行相关的理论知识。他们学会了如何构造可靠的异步电路、交换信号和消息、消除竞态条件、避免死锁以及防止时序依赖错误。他们设计了可以显式表示一个问题中自然存在的并行性的程序设计语言，从而使得编译器输出的大部分代码可以在多处理器计算机中显著受益。但遗憾的是，在 20 世纪 80 年代和 90 年代，因为业界对并行计算并不感兴趣，很多理论都沉寂了。2000 年前后，随着多核芯片的出现， 154

情况发生了变化：旧的原理在程序员中重新流行，它们具体体现在几个基本的模型中。

协作的顺序进程

这种模型通常用于操作系统的组织。一个并行系统由一组以未知速度并发运行的顺序进程组成。进程也称为线程，是在单一地址空间中执行一个程序时所产生的 CPU 状态序列。一个进程可以生出新的进程（“子进程”），进程之间通过交换信号和消息进行协作。

一个计算包含运行在共享地址空间的一个或多个进程。计算可以是终结的或非终结的。终结计算（terminating computation）是指计算内的所有进程都会结束，其输出就是留在共享内存中的值。非终结计算（nonterminating computation）是指其中至少有部分进程在循环执行，其输出是一些指定进程所产生的值序列。

进程之间的通信（无论是在同一个计算内还是不同的计算之间）是由显式的信号完成的，数据不会通过隐式通道交换。隐式通信（如在共享内存中留下数据）通常是错误的一大来源。程序员必须明确协作需求，并使用同步协议对它们进行显式的处理。典型的协作需求包括以下几种：

1）*竞态条件*。内存单元的值取决于并发进程向该单元写数据的顺序。

2）*互斥*。不同的进程不能同时执行临界区的代码。

3）*串行化*。代码段作为一个单元执行，不能与那些可能与其竞争的并发代码段交叉存取。

4）*同步*。直到接收到另一个进程的信号，进程才能越过指定点继续执行。

5）*集合点*。一组进程在指定点等待，直到组内所有成员都到达这个点，然后继续运行。

6）*消息传递*。进程之间相互发送消息。

7）*死锁预防或避免*。进程一定不能进入循环等待。循环等待指同组两个进程都暂停等待来自对方的信号。

8）*仲裁*。当两个信号几乎同时发生时，选取其中之一作为第一信号，另一个则作为
155 第二信号，这样就不会丢失任何一个信号。

功能系统

在这种模型中，并行系统包含一个共享内存的偏序任务集。每一个任务完成一个功能，将输入内存单元的值变换为输出内存单元的值。一个任务在其“初始化”到“完成”

的时间间隔内执行。任务在初始化（也称为“发射”）之后，可以读取输入并按照内部功能将其输出写入内存。当任务完成后，它就向偏序集中所有显式标记为其后继的任务发送已完成信号。一个任务只有当其接收到它的所有前驱任务发送的完成信号后才能初始化。这一模型有很多变体，包括 Petri 网、并行程序模式（Parallel Program Schemata）及数据流图（Dataflow Graphs）。诸如 APL、VAL 等编程语言就是用来表示这种模型中的并行计算。

事件驱动的系统

事件驱动的系统是一组进程的集合，当预定的事件发生时，就有信号会通知相关的进程。例如，网络管理进程等待两种类型的事件：一是来自互联网上的数据包到达，随后它们必须被分发到正确的接收方用户进程；二是来自用户进程的请求到达，随后其数据包被发送到互联网上。另一个例子是实时控制进程，例如医院中的病人监护系统，必须在指定时间内响应传感器的通知。这些系统对协作的顺序进程模型进行了扩展：允许一个等待进程能被一组信号中的任意信号唤醒，而不仅仅只是某个特定信号；同时还允许进程分时复用互斥区域。支持这种模型的编程语言从 20 世纪 70 年代开始就被广泛使用，包括管程（Brinch Hansen 1973，Hoare）以及面向对象的语言，如 Modula、Smalltalk、CLU 和 Occam。[2]

MapReduce 系统

MapReduce 系统最初是用于处理超大型数据库的并行查询和检索（Dean Ghemawat 2004）。这类系统在“大数据”方面的应用——即超大数据集的趋势和模式分析，已经非常成功。其核心思想是程序员必须将问题划分为成千上万个并行的小片段，这些小片段可以在互联网中分布的服务器上完成，而所有小片段的结果又能够快速地合并成原始大问题的解答。这一想法是由 Google 提出的，使得能够对分布在世界各地服务器上的 Web 数据 156
库进行大规模并行查询，这样每一个查询请求在 0.5 秒钟内就可以得到响应。Hadoop 就是一种实现 MapReduce 的开源语言。

由于篇幅的限制，我们不再进一步讨论事件驱动的系统和 MapReduce 系统。协作的顺序进程模型和功能系统模型足以揭示大型并行系统中协作的基本原理。

协作的顺序进程

这种模型从进程抽象（process abstraction）开始。第一个分时系统的设计者发明了这

种抽象，来处理多个程序之间可靠地切换 CPU 时的一些难题。[3] Edsger Dijkstra（1965，1968a，1968b）提出将操作系统实现为一组协作的顺序进程，这一想法被广泛地接受了。Tony Hoare（1978）将其整理为一个叫做 CSP（Cooperating Sequential Processes，协作的顺序进程）的模型，并导致了用于编写这种系统的 Occam 语言的出现。20 世纪 80 年代，Occam 在一些超级计算机中经常会用到。[4]

实现进程抽象涉及大量的低级操作系统行为。Dijkstra 用一个非常简单的用户接口隐藏了所有繁杂性。这个接口给出了创建和删除进程、挂起和恢复进程、进程之间交换信号等操作（见图 8.3）。

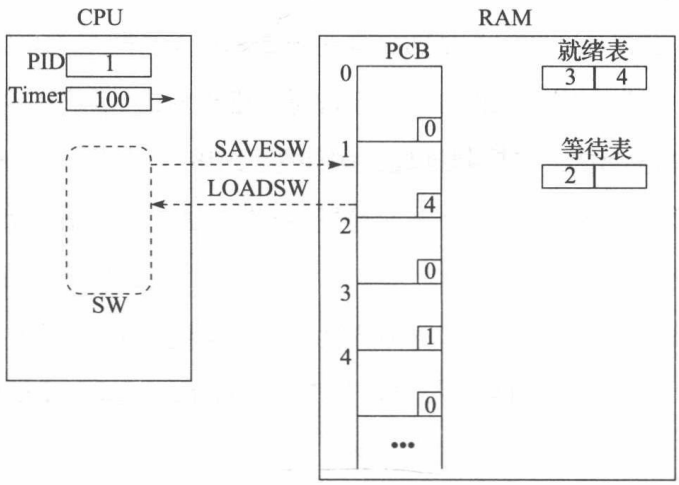


图 8.3 一组进程多路复用 CPU 的机制是建立在自动保存和载入进程状态字的基础上。所谓状态字（stateword，SW）是指属于某个进程的所有 CPU 寄存器的值——例如，程序计数器、栈指针、运算寄存器。操作系统维护一系列进程控制块（Process Control Block，PCB），每一个进程控制块包含了一个状态字的快照。PID（Process Identifier，进程标识符）寄存器保存了当前正在 CPU 上运行的进程的编号。图中进程 1 正在运行。SAVESW 指令将 PID 编号的进程的状态字复制到其 PCB 中。当 PID 中的进程编号是 i 时，LOADSW 指令将 PCB[i] 中保存的状态字复制回 CPU 中，以使进程 i 能从上一次被中断的地方继续运行。一个 TIMER 寄存器，由 LOADSW 指令初始化其时间片的值（此处是 100 毫秒），向下计数，当值变为 0 时触发一个超时中断。超时中断处理函数先执行 SAVESW，然后将就绪表头的下一个进程编号载入 PID，再执行 LOADSW。就绪表是准备运行的进程的队列，图中，其表头是进程 3，表尾是进程 4。PCB 中的小框表示表中的下一个进程，因此，完整的就绪表是（3,1,4）。还有一个等待表，其中的进程不能执行，因为要等待特定的事件，比如等待磁盘传输完成。图中进程 2 就在等待。一旦其所需事件发生，进程 2 就从等待表头移到就绪表尾。进程 0 被称为空闲进程，它自动排在就绪表和等待表的末尾，当没有其他进程准备好时就让它运行。每当没有准备好的用户进程时都让进程 0 运行，这一规则在就绪表为空时可以保护系统避免崩溃

并行进程将竞态条件问题带到软件中来了。大部分程序员并不习惯处理竞态条件，因为他们编写的程序是顺序的，运行在没有电路级别竞态条件的硬件上，并且不与其他程序交互。图 8.4 展示了 Alice 和 Bob 通过不同的 ATM 机同时访问一个共享银行账户时出现的问题。如果在不同 ATM 机上运行代码时在时间上有所交叉，就会产生各种不同的结果。该实现与 Alice 和 Bob 以为的各自交易互不干扰的印象不同。他们以为自己的交易是*串行的*（serialized），也就是两个进程总是先后执行，而并不会同时执行。串行化（serialization）是解决竞态条件的一种方式。

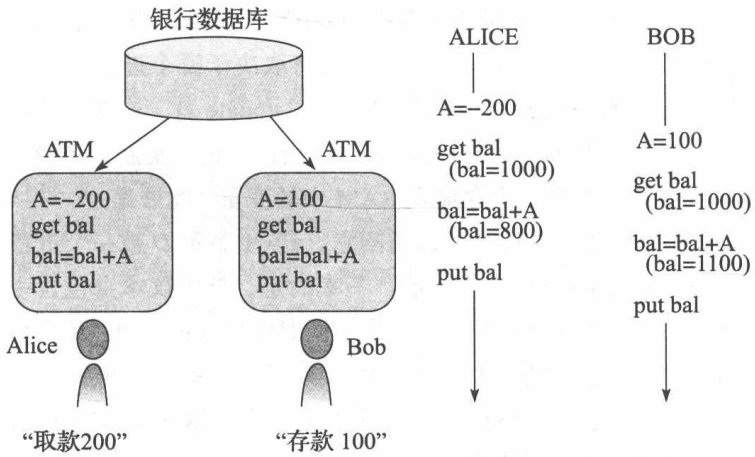


图 8.4　软件代码在并行执行时可能会产生竞态条件。左图显示了 Alice 和 Bob 试图从不同的 ATM 机同时在同一个账户上进行交易。每个 ATM 机运行一个简单的程序，可以从银行数据库中读取账户余额、根据交易额增加或减少账户余额、再将账户余额存回银行数据库。Alice 和 Bob 都认为他们的交易是不可分割的操作，不管谁先谁后，最终账户余额都将是 900 元。可是，如右图中的两条并行时间轴所示，如果两台 ATM 机几乎同时开始交易，并交错执行各自的指令，则可能出现错误。例如，Alice 先“查询余额”，Bob 再“查询余额”；然后 Alice 从账户余额的本地副本中扣除 200 元，Bob 向账户余额的本地副本中加上 100 元；接着 Alice“保存余额”，最后 Bob“保存余额”。这一序列的操作完成之后，最终的账户余额是错误的 1100 元。时序稍微转换一下，使 Alice 在 Bob 之后“保存余额”，则余额是 800 元，也是不对的

设计者试图用锁来解决这类问题，锁是一个保存值 1（加锁）或值 0（未加锁）的内存单元。锁被指派来保护一组特定的共享数据。一个进程在使用共享数据时加锁，使用完毕后解锁。想要使用该数据的任何进程先查看锁，只有当未加锁时才能继续执行。图 8.5 显示了这一过程是如何工作的。为了避免查看锁时的严重竞态条件，需要加入一条硬件辅助指令——“测试 - 置位”（test-and-set）指令。

157 ~ 159

图 8.5 中的加锁解决方案是有代价的——存在一个称为*忙等待*（busy waiting）的开销

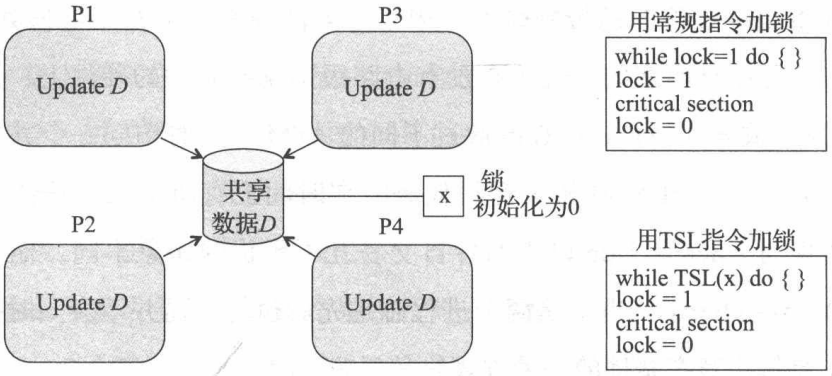


图 8.5 任何进程中试图更新数据的指令序列必须作为一个整体单元执行，以避免竞争错误。图中四个进程通过“update *D*”访问共享数据对象 *D*，“update *D*”可以是读写 *D* 的任意指令序列。若四个进程一起开始更新，*D* 最终的值将取决于哪个进程最后执行，并可能与它们以任何串行顺序执行所得到的 *D* 值都不同。像“update *D*”这样的必须作为一个整体单元执行的代码段，叫做临界区（critical section）。当任一进程更新共享数据时，对其加锁，就可以避免临界区竞争。锁是一个存储在 RAM 中的变量（此处是 *x*），并与共享数据 *D* 关联。*x* = 0 表示 *D* 未上锁，任一进程都可以访问它。*x* = 1 表示 *D* 被一个进程上锁，其他进程不可以访问。该锁协议见图中右上角所示。不幸的是，该协议本身包含一个错误，称为“锁竞争”：若两个进程在各自设置 lock = 1 之前同时读到 lock = 0，那么两个进程可以同时进入临界区。为了避免这个问题，大部分 CPU 实现了一个称为“测试 – 置位锁”（test-and-set lock）的硬件指令，缩写为“TSL *x*”，执行 TSL 指令将返回 *x* 的值并置位 *x* = 1。TSL 被实现在一个内存周期内完成，因此它不会被其他 TSL 中断

较大的问题。忙等待是指 CPU 循环测试锁状态，等待其解锁。忙等待会浪费大量 CPU 时间从而对分时系统造成严重影响。如果在进程等待锁时将其挂起，当锁释放时再将其恢复，则情况会好很多。

Edsger Dijkstra（1968a，1968b）发明了信号量（semaphore），很好地解决了这个问题。信号量是一个锁，其中包含了等待其解锁的进程的队列。任何想要获得加锁权限的进程，会立即从 CPU 的就绪表中拿出来，并放入信号量队列中。wait 和 signal 操作分别执行上锁和解锁，并管理队列（见图 8.6）。

信号量也为并行进程间很多其他同步问题提供了优雅的解决方案。每个同步会涉及三个方面：

1）一个或多个发送者进程。

2）一个或多个接收者进程。

3）发送者和接收者协作，使得每个发送者在到达指定点之前，没有接收者可以越过相应的指定点。

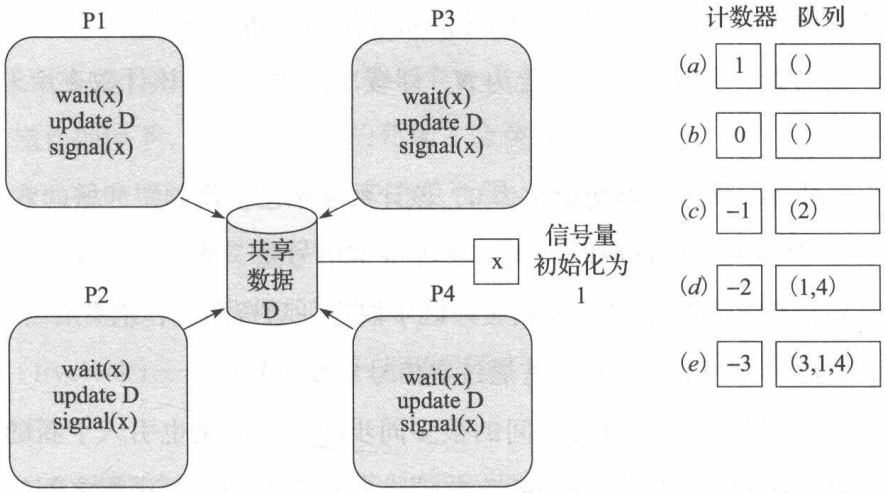


图 8.6　Edsger Dijkstra 发明了信号量作为临界区加锁的一种方式，避免了 TSL 方法中隐含的忙等待问题（图 8.5）。信号量从一个进程向另一个进程传递信号，并延迟接收方直到信号可用。它由一个计数器和一个队列组成。计数器初值是不需要等待就可以得到信号的个数。信号量 x 的 wait(x) 操作从计数器中减 1，如果结果为负，则将调用方放入队列并睡眠。注意负的计数器值的大小就是队列的长度。signal(x) 操作给计数器加 1，如果结果非正，就唤醒队列中的第一个进程。右图是信号量可能的状态的示例。*a* 是初始状态，计数器是 1，队列是空。第一个请求临界区的进程不需要等待直接通过。状态 *b* 表示某个进程在临界区中，没有进程在等待。状态 *c* 表示有一个进程（此处是进程 2）在等待。状态 *d* 表示两个进程在等待，队列是（1，4）。下一个 signal 操作将唤醒进程 1。状态 *e* 表示三个进程在等待，队列是（3，1，4），下一个 signal 将唤醒进程 3

发送者进程使用信号量发信号表明它已到达指定点。接收者进程使用同一个信号量在指定点停止并等待，直到接收到信号。对于这类信号，信号量提供了最简单的通道和排队机制。

一种常见的同步模式是进程从共享资源池中借用资源单元，之后再归还回去。内存页面就是这样的例子。当缺页时，虚拟内存管理器从空闲页面池提取一个未使用的页面，将其分配给缺页进程；之后，该页面通过替换算法归还页面池。资源池的使用是由信号量来保证其同步的：计数器记录了资源池中剩余资源的数量，wait 操作授权提取一项资源，signal 操作在资源归还资源池时发起通知。

采用信号量来解决的另一个常见同步问题是通过缓冲区将资源流从“生产者”进程传输到“消费者”进程。缓冲区是一个有限的存储区域，用来保存“在途中”的资源。这里的问题是要避免缓冲区上溢（overflow）或下溢（underflow）。上溢指生产者已经填满了缓冲区，而之前的资源在被消费者取走前又被新资源覆盖了。下溢是指消费者已经读取了全部缓冲区资源，在新的资源到来之前又再次读取了原来的资源。上溢会造成资源的丢失，

160
~
162

而下溢会造成资源的重复。解决该同步问题的协议使用了两个信号量：

1）“空”信号量，其计数器初始化为 N，即缓冲区的大小。该计数表示未使用的缓冲槽的数量。

2）“满”信号量，其计数器初始化为 0，该计数表示已使用的缓冲槽的数量。

要向缓冲区中插入一项资源时，生产者等待空信号量，将资源放置在下一个未使用的槽中，并通过满信号量发信号。要从缓冲区中取走一项资源时，消费者等待满信号量，从下一个存有资源的槽中取出资源，并通过空信号量发信号。

虽然信号量能够解决并行进程之间的很多同步问题，但是也引入了新的问题。最大的问题就是*死锁*（deadlock）。死锁是持有并请求共享资源的一组进程之间的一种循环等待状态。假定每个资源有一个锁表示它正在使用，死锁时，一组进程都停下来，每个进程持有一个锁，并等待组内的另一个进程释放锁。解除死锁的唯一出路就是杀死组内所有进程，释放它们的锁，并重新开始。

在前述 ATM 问题中可以看到死锁的例子。考虑 ATM 转账交易，这种交易是从付款账户中减去一个金额，并向收款账户中增加相同的金额。通过编程使 ATM 机在进行转账之前锁定两个账户，我们可以避免产生竞态条件。一旦两个账户被锁定，则当第一个交易正在使用这两个记录时，我们无须担心其他交易会访问它们。想一想，如果 Alice 和 Bob 同时开始转账，两个 ATM 机会发生什么（见表 8.1）。信号量 A 和 B 的计数器都初始化为 1。

表 8.1 设置一个死锁

ATM 1：	ATM 2：
“Alice(A) 向 Bob(B) 转账 100 元”	“Bob(B) 向 Alice(A) 转账 200 元”
wait(A)	wait(B)
wait(B)	wait(A)
A = A – 100	B = B – 200
B = B + 100	A = A + 200
signal(A)	signal(B)
signal(B)	signal(A)

假设两个 ATM 机同时开始，执行各自的第一组指令，使两个信号量都为 0。现在它们必然要失败，因为 ATM1 被挂起等待信号量 B，ATM2 被挂起等待信号量 A，它们都无法继续。

解决死锁有很多的方法，这里我们简要介绍最重要的三种[5]。第一种是让死锁发生，
163 然后杀死涉及的进程使系统脱离死锁。检测算法是简单建一个图，显示哪些进程持有锁、哪些进程正在请求锁，然后在该图中寻找环。环就表示循环等待。该方法通常不能令人满

意，不仅因为杀死死锁进程的代价很高，还因为该方法无法检测即将发生的不可避免的死锁。例如，上述的 ATM 机在完成各自第一个 wait 操作后并不会马上死锁，直到两者都在第二个 wait 操作处停下，该不可避免的死锁才会被检测到。

第二种方法旨在通过预先获取锁来避免死锁。在执行临界代码之前，进程先进入一个循环，在循环中获取它需要的所有锁后，然后才能继续执行。如果进程发现所需的某个锁已经上锁，它会释放已经获得的所有锁并重新开始。该协议可以避免死锁，其代价是可能造成活锁（livelock）——两个或多个进程在同步中无限循环，相互阻碍使得对方拿不到所有需要的锁。

第三种方式是强制按照固定的优先次序来获取锁，从而避免死锁。为所有的锁编号或指定字母数字名称。只有当新锁的编号或名称大于一个进程已经拿到的各个锁，该进程才可以请求这个新锁。有了这个约束，进程之间就不再会发生循环等待[6]。将此方法应用于前述 ATM 转账的例子中，转账程序首先将涉及的账户名按字母升序排列，然后按此顺序执行 wait 操作[7]。

功能系统

协作的顺序进程模型有一个隐含的假定：每个进程都是运行在一个指令速率和电路均由时钟控制的 CPU 上。并发是发生在进程之间，而不是进程内部。 164

受时钟控制的系统扩展性不佳，时钟节拍间隔必须超过信号在芯片中最长路径上传输的时间。尺寸更大的芯片意味着更慢的时钟。而在高密度的芯片上使用更快的时钟，会产生散热的问题。

发送就绪 - 确认信号可用于构造自定时电路，即没有时钟的电路。当 *A* 部件有一个对 *B* 部件的请求时，如要传输数据或要开始一个操作，*A* 先向 *B* 发送就绪信号。该就绪信号激活 *B*，当 *B* 处理完该请求时，*B* 向 *A* 返回一个确认信号。该过程结束后允许下一轮 *A* – *B* 交互开始，并可以无限重复。自定时电路的能耗要低得多，因为只有当数据被传输或变换时部件的状态才会发生改变。

最简单的（也是最早的）自定时系统模型是访问共享内存的任务网络。每个任务完成一项简单的功能。一些任务有先后顺序的约束，意味着只有当前驱任务都完成后它们才能开始执行。若任务之间不存在先后顺序的约束，则它们是*并发的*（concurrent）。当一个任务的所有前驱都完成后，该任务可以随时*激发*（fire）。当任务激发时，它从内存单元的指定输入集中读取数据，进行处理，再把结果数据写入内存单元的指定输出集中。一个任务

网络的激发次序就是一个与先后顺序的约束相一致、并按执行顺序排列的任务列表。那些有少量先后顺序约束、同时有大量并发任务的网络，其激发次序可以有很多种（如图 8.7 所示）。

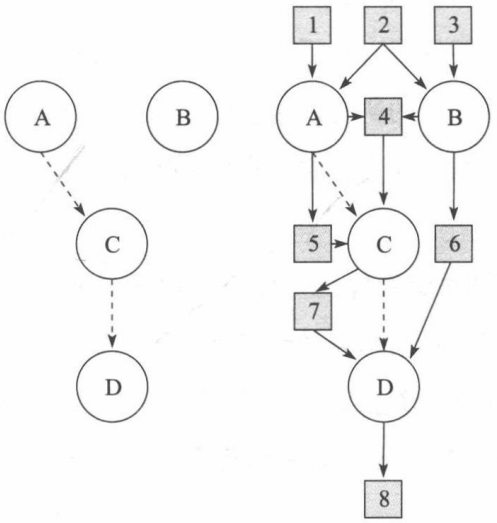


图 8.7 任务图将任务节点（用圆圈表示）实现的功能连接成网络。左图显示了四个任务 *A*、*B*、*C*、*D*，其中有两个用虚线箭头表示的先后顺序约束。任务可以按符合约束的任意顺序激发，并执行其功能。可能的激发次序是 *BACD*、*ABCD*、*ACBD*、*ACDB*。右图显示了每个任务从特定的内存单元（用灰色方框表示）中取得输入，并将输出写到另外的内存单元。例如，任务 *A* 从内存单元 1 和 2 中获得输入，再将它的结果写到内存单元 4 和 5 中；任务 *C* 从内存单元 4 和 5 中获得输入，再将其结果写到内存单元 7 中。不同的激发次序会产生不同的最终结果，最后都存放在内存单元 8 中。假定内存单元中的初始值和其编号相同，所有节点都执行加法操作，这样，激发次序 *BACD* 产生的结果是 11，*ABCD* 的结果是 13，*ACBD* 的结果是 11，*ACDB* 的结果是 12。这些不同是由两种竞态条件导致的：（1）输出 – 输出竞争，两个并发的任务将结果写入同一个内存单元。例如内存单元 4 中的值依赖于 *A* 和 *B* 的激发次序。（2）输入 – 输出竞争，一个并发任务写入的内存单元被另一个任务读取。例如，内存单元 6 就处于 *B* 和 *D* 之间的输入 – 输出竞争中，这会影响内存单元 8 中的值；而内存单元 4 则处于 *B* 和 *C* 之间的输入 – 输出竞争中，这会影响内存单元 7 中的值。为了避免竞争，不允许任何任务向并发任务读或写的内存单元中写数据

在一个没有竞态条件的网络中，我们期望对于该网络的每一个输入，只能从该网络得到唯一一个输出。网络的每一个输出值仅仅取决于内存中的初始值，而与激发次序无关。图 8.7 中的任务系统并没有实现一项“确定”的功能。

如果一个任务网络的总体行为是一项与其激发次序无关的功能，则称其为*确定性的*（determinate）。图 8.7 显示的是一个非确定性系统。即使各个任务实现了各自功能，也不能保证该任务系统作为一个整体实现了一项“确定”的功能。

对于天气预报、石油勘探、机翼设计等大型计算来说，确定性极其重要。组成计算的小任务大部分是并发的，它们可以并行运行，从而获得高性能。如果计算的结果取决于任务的某个确切激发次序，则无法保证得到的结果是正确的。我们怎样才能知道哪次执行是对的呢？我们可不希望因为设计机翼的超级计算机中或者飞行中控制机翼的航空电子系 165
统中不可预测的任务激发次序而使飞机从天上掉下来。

在调试中，非确定性也是一个严重的问题。如果一个错误无法被分离并重现，调试起来就非常困难，甚至是不可能完成的。依赖于并行任务激发次序的错误看起来像是随机的、间歇性的故障，无法被重现。就算它们能够被检测到，也很难在任务系统中定位产生故障的确切位置。早期的任务系统设计者把这种时间相关的问题称为“潜伏错误”(lurking bug)。

幸运的是，一种非常简单的约束就可以保证任务网络远离潜伏错误，并保证对于相同的输入，任务网络的每次执行都能得到相同的输出：*不允许任何任务向并发任务读或写的内存单元中写数据*。这就是并行系统的*确定性定理*[8]。

最后，我们再谈谈如何在没有时钟的自定时计算机上实现任务系统。MIT 的 Jack Dennis 和 David Misunas（1975）为此设计了数据流体系结构。尽管数据流体系结构还未被商业化，但它们是大规模自定时计算机的概念证明。比起当前的超级计算机，数据流体系结构在规模上有更好的可扩展性，而且能耗更低，因此，可能有一天它们在商业上会很具吸引力。

基本的数据流计算机由三个部分组成：执行单元、数据存储器、互联网络。执行单元包含的部件和传统 CPU 中的很像，例如加法器和乘法器。数据存储器包含任务系统图的表示，就像图 8.7 中的那样，但是要大很多。两类特殊类型的任务——选择器和迭代器，控制着全部子图的激活。互联网络是一个高度并行的网络，先将描述已激发任务的数据包传输到执行单元，这些已激发的任务会在那里被执行；然后再从执行单元中返回包含结果的数据包。因为数据包在数据存储器和执行单元之间往返耗时，在执行单元上执行一个任务的时间会比在传统 CPU 上慢一些，但由于可以大规模的并行，总体吞吐量可能会非常高。

任务的一种标准表示方法是一个包含输入、输出以及完成任务功能的操作代码等字段的数据项。例如：

(in1, p1, in2, p2, OP, out1, out2)

其中， 166 ~ 167

- in1 和 in2 是数据槽，其他任务在完成时会往其中插入数值。

- p1 和 p2 是标识位，分别指示 in1 和 in2 中的数值是否存在。
- OP 是操作的名称。
- out1 和 out2 是任务所需输入的地址，将接收 OP 的输出结果。

任务的执行周期如下：

- 当 p1 和 p2 都是 1 时，操作被激活。
- 数据存储器创建一个数据包，包含数值副本、操作代码、被激活操作结果的存放地址，并将该数据包发送给执行单元。
- 根据操作的类型，执行单元将数据包路由到相应的功能单元。
- 功能单元用数据包中提供的数值执行运算。
- 功能单元用数据包中提供的地址创建发往收件人的输出数据包。
- 输出数据包返回到数据存储器，其值被复制给目标任务的输入，并将相应的 p 位置为 1。

在这样的安排下，一旦数据包到达数据存储器和执行单元，动作就会被触发。

为数据流设计硬件并不是真正的瓶颈——现代体系结构已经包含了流水线、多核芯片和图形处理器，所有这些都依赖于数据流方法。真正的瓶颈在于算法设计和语言设计。

1976 年，普度大学的 John Rice 发表了一项关于数值计算性能的研究，如求解环绕飞翼的气流的向量场。1940 年以来，硬件速度已经提高了一百万倍（10^6），算法自身也提高了一百万倍，合起来就提高了 10^{12} 倍。换句话说，有一半的提高是来自于算法设计。

如今算法设计和 1976 年时一样重要。一个重要的区别是 John Rice 报告的性能提高是由数值计算专家创造的，并封装成标准的数值功能数学函数库。每天，程序员只是简单地使用函数库，并不需要考虑算法内部结构。

如今，由于使用多核计算机，世界各地的程序员必须设计更多的并行算法。然而，大部分程序员没有受过“并行思考”的训练。对于很多编程语言来说，多线程是一个非常
168 难用的附加部分。只有少数语言把并行功能作为其基础设计的一部分。为了使大部分程序更加并行化，程序员需要学会从一开始就考虑并行性，训练自己的思维去处理现实世界中存在的所有并行性，并使用新的语言表达自己的设计。

结论

看起来似乎很奇怪，一次只执行一步的计算机主宰了一个有着丰富并发活动的世界。并行系统应该是一个更好的选择，因为它们能调动数量庞大的处理器，所以可以提供比串

行计算更加显著的加速潜力。

这个好处是有代价的。并行性给程序员带来了新的挑战，其中最主要的就是竞态条件、同步和死锁。调试也变得更加困难。我们如何才能限制一个系统只做“正确”的执行，尤其是在可能的执行序列数量随系统规模呈指数级或更急剧的方式增长的时候？为了做到这一点，我们已经开发了大量的协议，包括使用信号量、在并发任务间共享内存以及等待竞争的资源等。

面向一组协作顺序进程的并行处理模型已经使用了很多年，它对传统的顺序进程模型进行了直观的扩展。该模型已被证明难以扩展到超大规模计算上。另一个重要的方法是功能系统模型，它面向大量的“就绪就激发”（fire when ready）型简单任务，这些任务共同实现对一个问题的求解。这种系统依赖于没有时钟的自定时电路。一个简单的结构化规则——不允许任何任务向被并发任务读或写的内存单元中写数据——足以保证任意规模的任务系统是确定性的。 169

第 9 章
Great Principles of Computing

排　队

确定性系统的不可预测行为源于驱动系统的工作负载缺乏确定性。

——Jeffrey P. Buzen

网络的最初想法是在很多人之间共享计算机、应用程序、软件及数据。

——Leonard Kleinrock

一家大型航空公司建立了一套计算机订票系统，并授权全球 1000 家代理在工作站上使用该系统出售他们飞机上的座位。一个位于秘密、安全位置的数据中心记录了所有航班、航线以及预订等信息。平均而言，每个代理每隔 60 秒向数据库发起一项作业。每项作业平均向数据中心的目录磁盘（directory disk）发出 10 个请求，来定位包含实际数据的其他磁盘。目录磁盘平均耗费 5 毫秒来处理每个请求，其繁忙时间达到 80%。

这套系统每小时可以处理多少个全球范围的作业？巴黎的一家代理体验到的平均响应时间是多少？如果有一种存储目录的新方法可以使每个交易的目录访问次数降低到 5 次，那么响应时间会有什么变化？如果代理的数量翻番，响应时间又会有什么变化？

这些都是关于“一个网络计算机系统响应大量竞争的自主进程的请求的能力”的典型问题——在前一章中我们称其为“竞争并行”（competitive parallelism）。我们需要预估在不同负载下网络系统的行为。在第 6 章“计算”中讨论过的算法分析（algorithms
171 analysis），远不足以用来回答这类问题。算法分析关注的是解决问题所需的 CPU 时间，而不关注其他必需的服务（例如网络连接、输入输出及存储访问）所带来的延迟。此外，这些服务带来的延迟不仅仅取决于服务器群的结构，还取决于其他竞争同一服务器群的进程所排成的队列。算法分析能回答一个独立进程的执行时间这样的问题，但它无法回答一个系统中多进程间相互竞争资源时的性能问题。

我们可以用排队分析（queueing analysis）来解决这类问题（Denning 1991a，1991b）。大多数人认为上面四个问题的答案取决于系统结构的细节——代理的工作站所在位置及其类型、每个工作站和数据中心之间的通信带宽、数据中心的磁盘数量和类型、磁盘的访问模式、数据中心的本地处理器和随机存取存储器、操作系统的类型、交易的类型，等等。确实，前两个问题——关于吞吐率（throughput）和响应时间（response time）——只利用

上述给出的信息就可以准确地回答。而对于第三和第四个问题，由于配置的改变，系统行为的合理估算可以通过以上可用的信息和几个可信的假设得出。

排队论遇上计算机科学

排队论是数学的一个分支，始于20世纪初期，用于预测排队中的等待延迟。它源于哥本哈根电话工程师A. K. Erlang（1909）的一项研究——预测自动电话交换系统中的呼损率（loss probability）。当调查呼叫者对电话系统的需求时——特别是到达（arrival）之间的时间（呼叫的次数）与服务时间（通话的总长度）——他发现到达时间和服务时间均呈指数分布。他发现一次到达间隔时间（interarrival time）超过 t 秒的概率是 $e^{-\lambda t}$，其中 $1/\lambda$ 是到达之间的平均间隔时间。相似地，他发现一个通话长度超过 t 秒的概率是 $e^{-\mu t}$，其中 $1/\mu$ 是平均的通话持续时间。到达和服务的指数分布假设，大大简化了Erlang的数学表达，并为电话交换提供了非常精确的模型。从此，在排队论中，字母 λ 和 μ 特指到达率和服务率。

过了几年，Erlang（1917）发表了一个用于预测电话交换系统呼损率的模型，该模型是出于对电话交换中的开销和复杂性的实际考虑。即使在小镇上，一个能够100%接收全 172
部可能的电话呼叫的中心也是极其昂贵的。Erlang的研究表明，一个更小的实惠的中心同样可以工作，只要镇上的居民能够接受当他们打电话时有很小的几率不被系统所接受。

Erlang的方法使用了俄罗斯数学家安德烈·马尔可夫在1906年发明的马尔可夫链（Markov chain）。马尔可夫链是一个随机过程，它包含一个状态序列，其中下一个状态的取值仅依赖于当前状态，而和之前的任何状态无关。当状态转换的间隔时间服从指数分布时，马尔可夫链可以相对容易地计算平衡状态分布，就是发现系统处于给定状态的长期概率（long-term probability）。Erlang使用马尔可夫链将电话交换系统的状态描述为同时进行的呼叫数量，以此就可以计算一次呼叫损失的概率。

利用马尔可夫链以及指数分布的到达和服务来找到排队系统中的平衡状态分布，这种方法是非常强大的。该方法使得很多领域的排队问题都能顺利解决，如交通、人流控制、库存控制、电话呼叫、制造业、医院管理、收费站管理等。

20世纪60年代，计算机设计者开始将排队论应用到计算机系统中，进行网络和分时系统的能力规划（capacity planning）。能力规划就是要计算需要多少资源，才能避免队列增长过长，并将响应时间限定在可接受的范围。Leonard Kleinrock（1964）在其博士论文中提出了预测通信网络中消息传输延迟的模型。网络数据包在经由各路由器最终到达目的地的途中经历了很多的排队延迟。Kleinrock的模型（1975，1976）已经在ARPANET的路由结构优化中得到应用。

对计算机系统做能力规划要难很多。组成计算机系统的网络节点是计算服务器，如CPU、文件服务器、输入－输出服务器等，而不是路由器。J. R. Jackson（1957）提出了第一个针对开放计算机系统网络的模型，其中所有服务器的服务时间都呈指数分布。开放网络（open network）接受每一个到达，其数量会不断变化。十年后，W. J. Gordon 和 G. F. Newell（1967）针对封闭网络（closed network）解决了同样的问题。一个封闭网络包含固定数量的作业（job）。封闭网络很普遍。例如，所有用户都在系统边界内的系统，限制网络中数据包数量的网络协议，或者在一个作业离开时外部调度器会加入另一个新作业的系统，这些都是封
173 闭网络。Jackson-Gordon-Newell 网络的数学结构十分适合各种计算机系统。遗憾的是，对他们的公式进行求解的已知算法的复杂性是指数级的，除了一些小型系统外，对于所有其他系统，该解决方案都有不可逾越的难度。因此，Jackson-Gordon-Newell 模型几乎没有实用价值。

这个状况在 1973 年有了变化。这一年，Jeffrey Buzen（1973）发现了 Gordon-Newell 方案中一个隐含的结构，并给出了一个可以在二次方阶时间内求解的算法，而不是原来的指数阶时间。Buzen 的算法开启了一个新时代，使得分析师使用便携计算器就能解决很多领域中系统的吞吐率和响应时间的问题。同时也引发了封闭网络模型预测与实际系统比较的实验研究热潮，其结果也是惊人的一致——吞吐率的偏差在测量值的 5% 以内，而响应时间则在 25% 以内。

两年以后，Forest Baskett、Mani Chandy、Richard Muntz 和 Fernando Palacios（1975）发表了一个定理（就是现在大家所熟知的 BCMP 定理），它将 Jackson-Gordon-Newell 网络泛化为任意路由、服务分布以及多类作业，只要所有服务器在它们的队列中使用四个基本调度规则中的任意一个。这四个调度规则是先进先出（FIFO）、共享处理器、纯延迟（pure delay）、抢占式后来先服务。Buzen 的算法也随该模型进行了泛化，使得实践中可能遇到的几乎任何网络都可以拥有快速的计算方案。Martin Reiser 和 Steve Lavenberg（1980）很快发现了一个新算法，可以直接计算吞吐率、响应时间、队列长度的平均值，而 Buzen 的算法并不能直接计算这些平均值。他们的“均值算法”已经成为排队模型计算的标准。

这些模型的成功暴露了一个矛盾。计算机系统并不符合传统随机模型的关键假设，特别是平衡的服务分布和指数的服务分布。然而，这些模型可以很好地预测它们的吞吐率和响应时间。例如，计算机系统并不会表现出平衡性——它们的性能随着负载而变化，而不同时期负载总是在不断地变化。性能分析人员发现，更弱的流量平衡（flow balance）假设，即到达的数量和完成的数量相等，可以导出与随机平衡相同的数学公式（Buzen 1976，Denning & Buzen 1977，1978）。在很多计算机系统的很多时间间隔内，都非常接近流量平衡。

计算机系统网络的另一个异常方面，是服务的分布通常明显不是呈指数分布的。性

能分析人员发现，更弱的服务器独立（server independence）假设，即网络中服务器的输出率（output rate）只取决于本地队列长度而与任何其他服务器的队列无关，可以给出与随机模型相同的数学公式（Buzen 1976 , Denning & Buzen 1977, 1978 ）。在很多计算机系统的很多时间间隔内，服务器独立也是大体成立的。 174

结论是，排队论的经典假设可以被流量平衡和服务器独立这样的简单假设取代，同时仍然可以产生与经典排队网络模型相同的公式。这些简单的假设与很多计算机系统中观察到的很接近。这就是为什么排队论能很好地用于计算机系统中。[1]

用模型计算和预测

排队模型可以根据负载参数（如网络中每台服务器的平均服务时间和访问数），推导出表示性能指标（如吞吐率、响应时间或拥塞）值的公式。通过隐藏系统中的很多细节，模型提供的计算性能指标的方法比直接测量快很多。分析人员通过比较模型的计算结果和实际运行系统中测量的指标值来验证模型，已验证的模型通常能避免直接测量（如图 9.1 所示）。

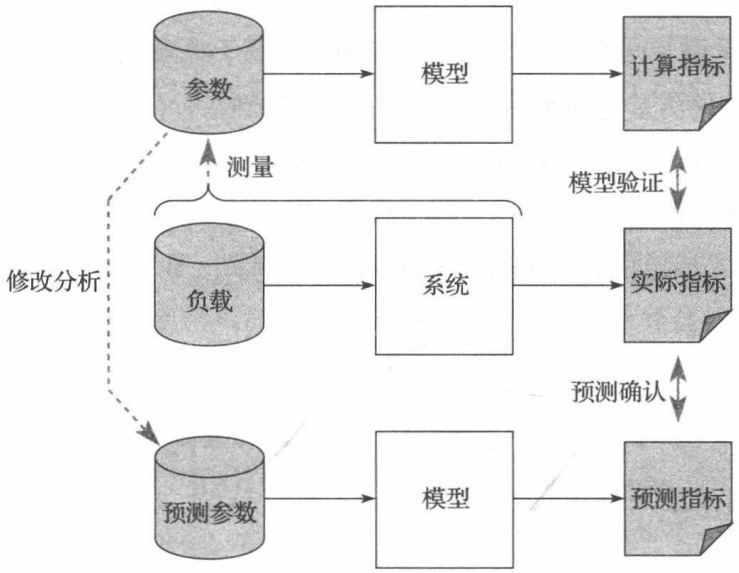


图 9.1　已验证的模型使得系统性能预测变得可行。系统和其负载可以被测量以获得如吞吐率和响应时间等指标的实际值（图的中部）。模型是一个算法，其输入是负载和系统的测量参数，其输出是各指标的计算值（图的上部）。验证时，将模型反复地与不同的系统相比较，以确认该模型可以很好地计算指标。验证后的模型可用于预测。为了预测未来的指标值，分析人员会通过询问什么会改变及如何改变来修改参数。之后该模型就可以用来计算预测的指标（图的下部）。因为模型已验证，预测中的错误最有可能是修改分析中的错误导致的

到目前为止，模型最常见的用途是能力规划。设计者利用模型来评估一个未来的系统能否满足吞吐率和响应时间的要求。例如，它们可以计算在要求的吞吐率和响应时间范

围内系统可以承受的最大负载，或者在遇到瓶颈时需要增加多少处理能力才能达到目标。它们还可以评估规划的结构改变是否会有效，例如一个规划的控制系统是否能防止性能颠簸。

服务器、作业、网络和规则

计算机网络是一组互联的服务器。服务器可以是工作站、磁盘、处理器、数据库、打印机、显示器以及任何其他可以执行计算任务的设备。每个服务器从其他服务器接收包含指定任务的消息，并将它们排成队列。一个典型的消息可能是让服务器运行一个计算密集型程序，或者执行一个输入－输出事务，或者访问数据库。作业是向网络提交的一个特定任务序列。当服务器完成了作业中的某个任务，会将其从任务队列中删除，并发送消息给另一个服务器，请求它执行同一个作业中的下一个任务。这样，作业在网络中流动，每次访问一个服务器。

175 ~ 176

我们的目标是，对于这样一个给定了工作负载（作业）和系统（互联的服务器）参数的计算机网络，预测其性能指标（如吞吐率和响应时间）。

服务器的测量通常是在一个确定的观察期中进行，该观察期持续 T 秒。我们来看看在观察期中，该如何测量各种参数并找到将它们关联到指标的公式。

通过统计向外发出的消息数量，并测量服务器队列的非空时间，可以很容易测量出服务器的输出率 X、平均服务时间 S 以及利用率 U。这三个实测量满足关系 $U = SX$，被称为*利用率法则*（utilization law）（如图 9.2 所示）。

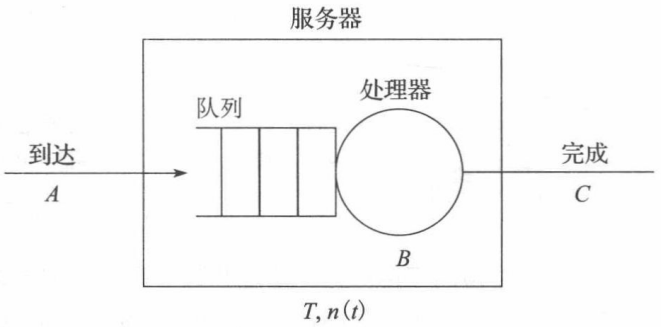


图 9.2 一个单服务器系统由一个处理器和一个等待服务的作业队列（存储区域）组成。在一个 T 秒的观察期内，A 个作业到达并加入到队列中，C 个作业完成服务并离开队列。系统的状态 $n(t)$ 是在 t 时刻系统内服务中或等待服务的作业数量。作业到达时状态增加，作业完成时状态减少。当 $n(t) > 0$ 时，系统繁忙，当 $n(t) = 0$ 时，系统空闲。计时器 B 记录了处理器的总的繁忙时间。服务器的利用率 $U = B/T$，完成率 $X = C/T$，已完成作业的平均服务时间 $S = B/C$。因为 $B/T = (C/T)(B/C)$，我们得到利用率法则：$U = SX$。因为 U 不可能比 1 大，所以 X 不可能大于 $1/S$，这意味着完成率不能快于每个平均服务时间完成一个作业

177

类似地，通过测量由队列任务累加的“作业时间”，可以很容易得到平均队列长度 Q 和平均响应时间 R：这些量满足关系 $Q = RX$，被称为利特尔法则（Little's law，1961）（如图 9.3 所示）[2]。

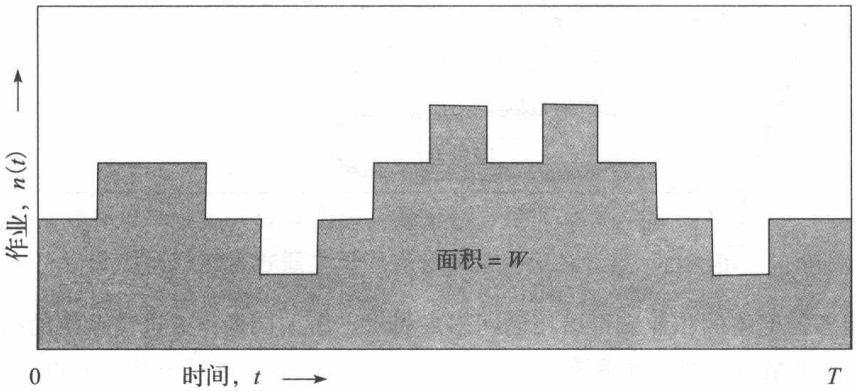


图 9.3　服务器的平均队列长度和平均响应时间可以通过计算观察期内 $n(t)$ 曲线下面的阴影面积 W 得出。W 是系统中所有作业累加的作业 - 秒（job-seconds）数。例如，一个在队列中等待了 10 秒之后收到了 2 秒服务的作业产生了 12 个作业 - 秒。W 相当于房地产代理计算房租时使用的“平方英尺 - 月”，或者项目经理计算人工费时使用的“人 - 月”。服务器的平均作业数是 $Q = W/T$，即曲线的平均高度。平均响应时间是 $R = W/C$，即将 W 分摊到所有完成的作业。因为 $W/T = (C/T)(W/C)$，所以我们有了利特尔法则：$Q = RX$。平均服务时间 S 和平均响应时间 R 不同，R 包括排队延迟和服务时间

利特尔法则是一个非常重要的公式，适用于任何满足下面条件的情况：一个可以容纳项目（item）的黑盒子，盒中有一定的响应时间，有一个项目流穿过这个盒子。考虑一个简单的例子：一家餐馆的主人出售高档酒，每年平均每天销售二十瓶高档酒。她想在供应给顾客前使每瓶酒都陈化十年，她需要一个多大的酒窖？显然，每年她必须能够提取出 $7300 = 365 \times 20$ 瓶酒，因此 10 年的陈酒将需要容纳 73 000 瓶的酒窖。她的计算实际上使用了利特尔法则：

$$73\,000 = Q = RX = (10\text{ 年})(7300\text{ 瓶 / 年})$$

组成作业的任务可以被看做是作业对网络中各服务器的一个访问序列。每个作业中对某个服务器 i 的平均访问次数称为对该服务器的访问率（visit ratio），记作 V_i；该服务器的输出率 X_i 和整个系统网络的输出率 X 满足关系 $X_i = V_i X$，这就是限制流量法则（forced-flow law）（如图 9.4 所示）。这一重要的法则表明，只要知道所有的访问率以及任何一个服务器的输出率就足以确定所有其他服务器的输出率及整个系统网络本身的输出率。该法则与服务器之间如何互联无关，任何两个访问率相同的系统网络具有相同的流量。

178

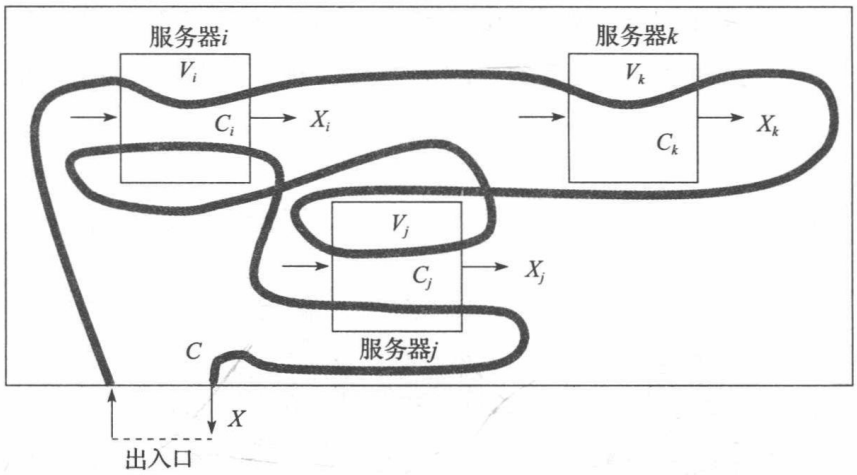


图 9.4 穿过服务器网络的作业流可以用访问率来表示。出入口是作业进出系统的指定点。如果系统中的作业数可以变化则系统是开放的，如果作业数固定则系统是封闭的（加入跨过出入口的虚线）。灰色路径显示了一个作业是如何访问服务器的；这里，该作业访问了服务器 k 一次，访问了服务器 i 和 j 各两次。在一个时间长度为 T 的观察期内，在各服务器上测量各自的完成数 C_i、C_j 和 C_k，在系统出入口处测量总的完成数 C。服务器 i 中每个作业的平均任务数是 $V_i = C_i/C$，V_i 被称为*访问率*，这是因为每个任务都被看作是作业对服务器的一次“访问”。恒等式 $C_i/T = (C_i/C)(C/T)$ 就化为*限制流量法则*：$X_i = V_i X$。该法则表示系统中一个点的任务流决定了各处的任务流

作为流量平衡的一个条件，假设服务器的输入和输出流是相同的，这可以使分析变得更简单。平衡的流量就是吞吐率。上述的各基本量和法则并不依赖于也不意味着流量平衡。尽管如此，对实际系统来说，流量平衡仍是一个非常接近真实情况的假设。由于同时存在系统中的作业数会有一个限制 N，因此在观察期内的初态和终态之间可能出现的最大
179 差异就是 N；只要每台服务器的完成数都比 N 大，由流量平衡假设而引起的误差就可以忽略。

当服务器网络从有限的 N 个用户那里接收对其发出的全部请求，在提交一个新事务之前，每个用户平均延迟 Z 秒，网络中每个请求的响应时间满足*响应时间法则*（response-time law）$R = N/X - Z$（如图 9.5 所示），这对流量平衡而言是非常精确的。

这些公式足以回答本章开头提出的关于航空订票网的吞吐率和响应时间问题。我们可以把之前给出的目录磁盘相关的信息表示为 V_i =10，S_i = 0.005 秒，U_i = 0.8。结合限制流量法则和利用率法则，可以得出系统的总吞吐率：

$$X = U_i/(V_i S_i) = 0.8/(10 \times 0.005) = 16 \text{ 个作业每秒}$$

也即每小时达到 57 600 个作业。1000 个代理中的任何一个体验到的响应时间是：

$$R = N/X - Z = 1000/16 - 60 = 2.5 \text{ 秒}$$

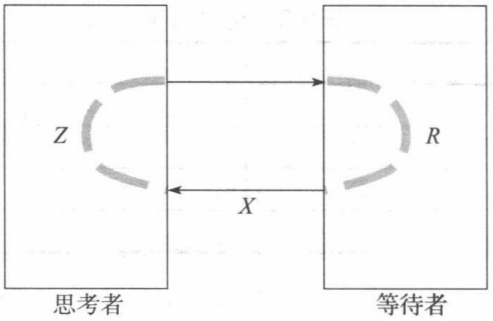


图 9.5　计算机系统的用户交替处于“思考”阶段和“等待”系统响应阶段。封闭系统的用户总数——思考者和等待者——固定为 N。每个事务的平均响应时间是 R，平均思考时间是 Z。将利特尔法则应用到思考者方框，可以得到其平均队列长度 $Q_1 = ZX$。将利特尔法则应用到等待者方框，可以得到其平均队列长度 $Q_2 = RX$。因为 Q_1 和 Q_2 的和是定值 N，则 $N = (R + Z)X$。求解 R，可以得到响应时间法则：$R = N/X - Z$。该公式假设流量平衡

179
~
180

我们并不需要去直接测量吞吐率和响应时间，因为通过一些简单的测量值就可将它们准确地计算出来。如果将这些计算结果与相同观察期内的实际系统相比较（如图 9.1 所示），它们会是很准确的。然而，在未来的一段时期内，它们可能并不准确，因为未来所需的参数与现在的参数可能不同了。

我们刚刚结束了排队网络中最困难的部分——学习了服务器的 6 个基本原始量、5 个指标和 4 条法则。我们把它们归纳在表 9.1、表 9.2 和表 9.3 中。

表 9.1　服务器的基本原始量

符号	描述
T	观察期
A	到达数
B	服务器总的繁忙时间
C	完成数
$n(t)$	在 t 时刻系统内服务中或等待服务的作业数，也称为队列长度
W	观察期内函数 $n(t)$ 曲线下的面积

表 9.2　服务器的基本性能指标

符号	定义	描述
X	C/T	完成率（作业数 / 秒）
U	B/T	服务器利用率（忙时的比例）
S	B/C	平均服务时间（秒）
Q	W/T	平均队列长度（作业数）
R	W/C	每个作业的平均响应时间（秒）
V_i	C_i/C	服务器 i 的访问率

表 9.3 运算法则

法则	公式
利用率法则	$U = SX$
利特尔法则	$Q = RX$
限制流量法则	$X_i = V_i X$
响应时间法则	$R = N/X - Z$

瓶颈

关于航空订票系统后面两个问题，是问在未来的观察期内，如果条件发生变化，其响应时间该如何估算——例如，目录磁盘的访问率减少，或者代理数增加。因为运算法则
181 只处理同一观察期内各观察量之间的关系，它们并不足以来做预测。我们必须引入额外的预测假设（forecasting assumption）。根据过去观察期测量到的参数值，这些假设可以推断出未来观察期所需的参数值，然后运算法则就可以用于计算未来时期内预期的响应时间（参见图 9.1）。

一种常见的预测假设是：除非特别说明，对于不同服务器的需求（即每个服务器的访问率）V_i，在未来观察期内其值和基准观察期内的相等。类似地，除非特别说明，主要取决于设备机械和电气特性的平均服务时间 S_i 也保持不变。而当其中有任何参数改变时，利用率、吞吐率和响应时间都将会发生变化。

回到前面的问题：当使用一个新的磁盘索引算法后，目录磁盘的访问率从 10 下降到 5，这会导致什么变化？这取决于目录磁盘是否是系统的瓶颈。如果不是，另外的一些服务器就会是系统瓶颈，大部分作业将会在那里排队，其利用率将会接近 100%。这种情况下，降低对目录磁盘的请求只会对瓶颈磁盘的利用率和吞吐率带来轻微的影响；限制流量法则告诉我们，整个网络的总体吞吐率和响应时间并不会因此而改变。

如果目录磁盘是瓶颈，基于限制流量法则，我们可以推测，将对其的请求降低一半会使系统吞吐率翻番。但是该推测产生了一个可笑的结果：对于上面给定的数据，通过公式算出的响应时间是 -28.75 秒。负的响应时间明显是荒谬的——意味着在问题提出之前就有了答案——这表明当目录磁盘上的请求降低一半后，它就不会是瓶颈了，即使它原先是瓶颈。就已有的信息和已给出的预测假设，我们所能说的是，对目录磁盘的请求降低一
182 半将会使响应时间从 2.5 秒降低到某个较小的非零非负值。如果 2.5 秒的响应时间已经是可以接受的，则这个改进的目录检索策略将不具有成本效益。

回到前面的最后一个问题：如果代理的数量翻番，响应时间会有何变化。同样，我们受限于缺少其他服务器的信息。若目录磁盘是瓶颈，那么代理数翻番可能会使其利用率

达到 100%，由此可以得到吞吐率的饱和值：

$$X = 1/(V_i S_i) = 1/(10 \times 0.005) = 20 \text{ 个事务每秒}$$

应用响应时间法则，我们可以得到此时的响应时间：

$$R = N/X - Z = 2000/20 - 60 = 40 \text{ 秒}$$

如果目录磁盘不是瓶颈，那其他某些服务器将会有更小的饱和吞吐率，使得响应时间超过 40 秒。因此，代理数翻番将会使得响应时间长得令人难以接受。

这个例子表明，在预测吞吐率和响应时间时，瓶颈分析是一个不可避免的主题（如图 9.6 所示）。假设所有服务器的访问率和平均服务时间都是已知的，并且不随 N 变化。每一台服务器会产生潜在的瓶颈，使得系统的吞吐率被限制到 $1/(V_i S_i)$，并且响应时间达到 $NV_i S_i - Z$ 的下界。显然 $V_i S_i$ 值最大的服务器具有吞吐率的最小上界，也就是真正的瓶颈所在。$V_i S_i$ 的乘积就是我们确定网络瓶颈所需要的全部信息。

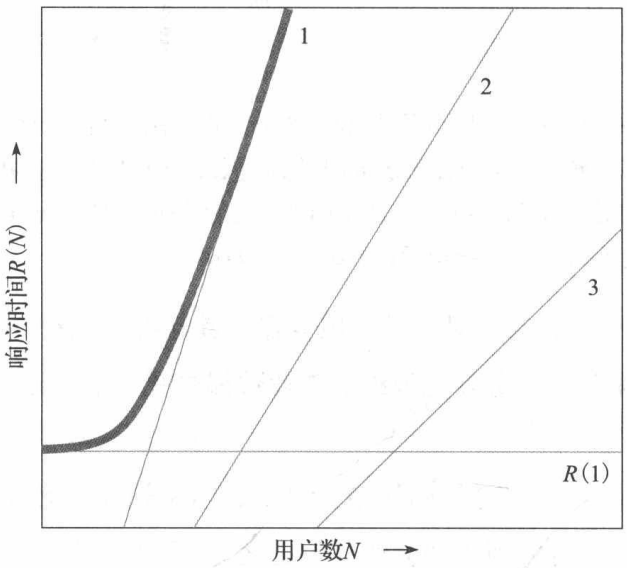


图 9.6　瓶颈分析展示了响应时间随 N 变化的函数。当 $N = 1$ 时，单一用户的作业并不会遇到由其他事务引起的排队延迟，因此 $R(1) = V_1S_1 + \cdots + V_KS_K$，其中 K 是服务器的数量。结合利用率法则和限制流量法则，因为 $U_i < 1$，所以 $X = X_i/V_i = U_i/(V_i S_i) < 1/(V_i S_i)$。因此，对所有的 i，$R(N) > NV_i S_i - Z$。当 N 较大时，这些关系确定的每一条线（如 1、2、3）都是 $R(N)$ 潜在的渐近线。实际的渐近线（粗线）是由最大的那根潜在渐近线（线 1）确定。假设服务器被编号，使得 V_1S_1 是最大的，V_2S_2 是第二大的，等等，则服务器 1 是瓶颈，并且 $R(N) > NV_1S_1 - Z$。瓶颈分析假定 $V_i S_i$ 的乘积不随 N 变化

瓶颈分析是一种简单但非常有效的计算吞吐率和响应时间的限制渐近线（limiting asymptote）的方法。瓶颈所在始终是请求总量 $(V_i S_i)$ 最大的服务器。不管 N 值是大还是小，下界 $R(1)$ 和 $NV_1S_1 - Z$ 可以相当接近。这些渐近线和实际的 $R(N)$ 之间的最大的误差

发生在渐近线与横轴相交处，即 $N = Z/(V_1S_1)$。为了获得某个负载下更精确的结果，我们必须借助于接下来所介绍的计算算法。

平衡方程

平衡方程是对排队系统进行更精确分析的必备工具，从 Erlang（1917）以来的排队论理论科学家都用过平衡方程。我们将就单一服务器系统来介绍这个方法。在后面的讨论
183 中请参考图 9.7。

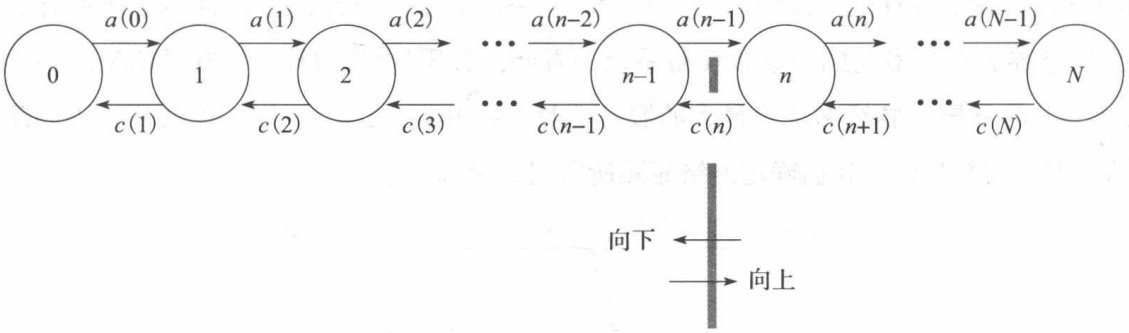


图 9.7 对单一服务器系统的状态空间分析引出了平衡方程。图中是可容纳 N 个作业的单一服务器系统的状态空间。$a(n)$ 是系统在状态 n 时的到达数，$c(n)$ 是系统在状态 n 时的完成数。平衡意味着系统的初始状态和结束状态是相同的，也就是说从状态 $n-1$ 向上的转换数和向下到达状态 $n-1$ 的转换数相同。换句话说，$a(n-1) = c(n)$

第一步是定义系统的状态。对于单一服务器，其状态是 $n(t)$，指服务器上的作业数。对于大部分真实系统来说，服务器能容纳的作业数都会有一个上界 N。因此，状态分别是 0，1，2，…，N。

第二步是定义状态之间的允许的转换。例如，当系统在状态 3 时，一个到达将会使其状态变为 4，而两个同时的到达则会使其状态变为 5。对大部分真实系统来说，到达和完成必然是不同的事件。我们将同时的到达建模为时间上非常近的一系列到达。因此，我们能看到的状态变迁就是：在到达时的“向上加 1”以及在完成时的“向下减 1”；同时服
184 从约束，即状态 N 时不能向上，状态 0 时不能向下。

第三步是根据流量平衡写一个平衡方程。流量平衡意味着在一个观察期内系统的结束状态和初始状态相同，也就是 $n(0) = n(T)$。在这种情况下，从任何状态 $n-1$ 向上的转换数与向下到达状态 $n-1$ 的转换数是相等的：

$$a(n-1) = c(n)$$

这与下面的方程等价：

$$\frac{a(n-1)}{T(n-1)}\frac{T(n-1)}{T} = \frac{c(n)}{T(n)}\frac{T(n)}{T}$$

左边的第一项定义为 $\lambda(n-1)$，这是系统处于状态 $n-1$ 时的到达率。右边的第一项定义为 $\mu(n)$，这是系统处于状态 n 时的完成率。左边和右边的第二项都是 $p(n)$ 的实例，而 $p(n)$ 是系统处于状态 n 的时间比例。因此，我们有平衡方程：

$$\lambda(n-1)p(n-1) = \mu(n)p(n)$$

或

$$p(n) = p(n-1)\frac{\lambda(n-1)}{\mu(n)}$$

所有 $p(n)$ 实例的和为 1，结合这一点，该平衡方程可以在电子表格上很容易地解出来[3]。该方程表明，到达率和完成率完全决定了一个流平衡系统（flow-balanced system）处于任一状态的时间比例。 185

一旦得到 $p(n)$ 的解，我们可以看到，服务器的利用率可以很简单地算出来：$U = 1 - p(0)$。而服务器的完成率，我们可以用每个状态 n 的完成率的加权平均来计算：

$$X = \sum_{n=1}^{N} \mu(n)p(n)$$

我们可以用处于状态 n 的概率的期望值来计算平均队列长度：

$$Q = \sum_{n=1}^{N} np(n)$$

响应时间则可以用利特尔法则来计算：$R = Q/X$。

这些计算与标准排队论（standard queueing theory）在数学上是完全相同的，但是依赖于不同的假设。在标准排队论中，$p(n)$ 是观察到系统处于状态 n 的长期平衡概率（long-term equilibrium probability）。而这里，$p(n)$ 是系统被观察到处于状态 n 的时间比例。如果系统是流平衡的，则该值是准确的；与 N 相比，如果系统有很多到达或完成，则该值是一个好的近似值。这就是为什么标准排队论的平衡公式在真实计算机系统中也可以很好地工作的原因。

我们以三个不同的计算机系统配置为例来说明该方法。

ATM

银行的 ATM 看起来是一个单一的服务器，其队列长度不超过 N，即 ATM 等待区域的大小。不管队列有多长，潜在的用户数很大使得总的到达率是稳定的。在这种情况下，到达率是 $\lambda(n) = \lambda$；服务率是 μ，因为每次只有一个用户被服务。则平衡方程为：

$$p(n) = p(n-1)\frac{\lambda}{\mu}$$

图 9.8 展示了使用电子表格求解该平衡方程的结果。

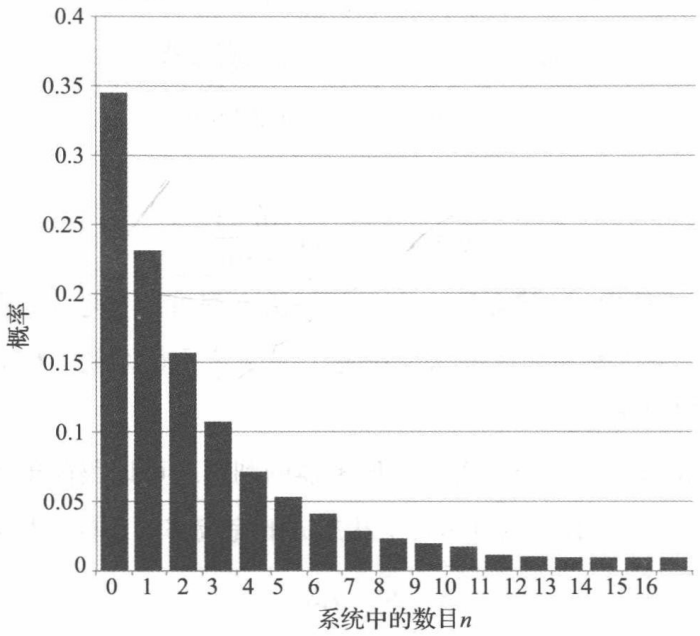


图 9.8 使用电子表格，可以得到 ATM 的最大队列长度 $N = 16$ 且 $r = \lambda/\mu$ 取不同的值时，$p(n)$ 的分布图。这里的图是 $r = 2/3$ 时的 $p(n)$ 的分布。因为 $\lambda<\mu$，因此该图严重向小队列偏置：队列长度小于等于 2 的时间占 70%，空闲时间占 33%。当到达率和完成率的值反过来（$r = 1.5$），会形成一幅镜像的分布图（图未给出），表明系统在 70% 的时间里有 14 个或以上的用户在排队，且没有空闲时间。当到达率和完成率相同（$r = 1$），所有条形高度相同；所有 16 种队列长度的可能性相同，系统有 6% 的空闲时间

电话交换机

这是 A. K. Erlang 在 1917 年研究的问题。电话交换机的状态 n 是进行中的通话数量。设备的局限将同时通话数限制为 N。当状态为 N 时，试图拨打电话的用户将会被拒绝。因为用户数量很大，在所有状态下到达率都是稳定的 λ。一次通话的平均持续时间是 $1/\mu$。在状态 n 时，有 n 个通话在同时进行，此时的组合完成率是 $n\mu$。得到的平衡方程为：

$$p(n) = p(n-1)\frac{\lambda}{n\mu}$$

Erlang 的设计中的问题是如何选择 N（即交换能力），使得用户被拒绝的可能性低到可以接受。例如，把可接受的呼损率设置为 0.001，这意味着我们要找到一个 N 值使得 $p(N) < 0.001$。图 9.9 展示了使用电子表格求解该平衡方程的结果。

186
~
187

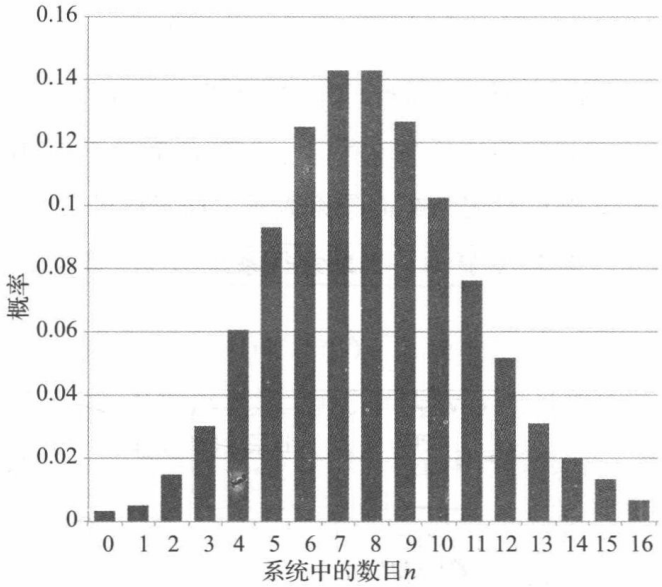


图 9.9　使用电子表格，得到了电话交换问题中当通话数最大值 $N = 16$ 且 $r = \lambda/\mu = 1$ 时的 $p(n)$ 的分布图。该图表明交换机在半满状态时有很强的偏好性。该交换机很少空闲并具有较小的呼损率 $p(16) = 0.005$

分时系统

MIT 的 Allan Scherr(1965) 为兼容分时系统（Compatible Time-Sharing System，CTSS）建立了一个性能模型，可以以惊人的准确度来预测系统的吞吐率和响应时间。他使用了排队论中一个叫做机器修理工（machine repairman）的模型，其按下述方式工作：在车间里，一个修理工为 N 台机器服务，这些机器各自按 λ 的比率发生故障，每修复一台机器平均需要 $1/\mu$ 的时间。系统的状态是排队等待修复的故障机器数量。将该模型转换到分时系统：N 表示用户数量，修理工变成平均服务时间是 $1/\mu$ 的 CPU，机器则变成各自需要思考 $1/\lambda$ 时间的用户。在这种情况下，对于处在状态 n 的 CPU，其到达率是 $\lambda(n) = (N - n)\lambda$，因为当 n 个用户在等待 CPU 时，$N - n$ 个用户在思考。因为同时只能处理一个作业，状态 n 的完成率是 $\mu(n) = \mu$。此时的平衡方程是：

$$p(n) = p(n-1)\frac{(N-n+1)\lambda}{\mu}$$

188

Scherr 通过向 CTSS 操作系统的内核中插入探针来捕获作业开始和结束事件记录、对 CPU 队列长度和响应时间进行采样，来验证他的模型。他将获得的数据与模型计算的结果相比较，确认模型很好地预测了 CTSS 系统的吞吐率和响应时间。

Scherr 的结果出乎很多人的预料，他们不相信这么一个简单的模型能够很好地估计复杂分时系统的吞吐率和响应时间。

用模型来计算

作为建模过程中的最后步骤，我们希望将模型扩展到任何排队网络。像之前一样，我们需要定义系统的状态和它们的平衡方程。一个有 K 个服务器的网络，其状态比单一服务器要更复杂。一个状态可以用一个向量来表示，向量中的分量表示 K 个服务器中每一个服务器的作业数量。图 9.10 显示了机票预订系统的一个模型，以及当 $N = 2$ 时，其可能的 10 个状态。

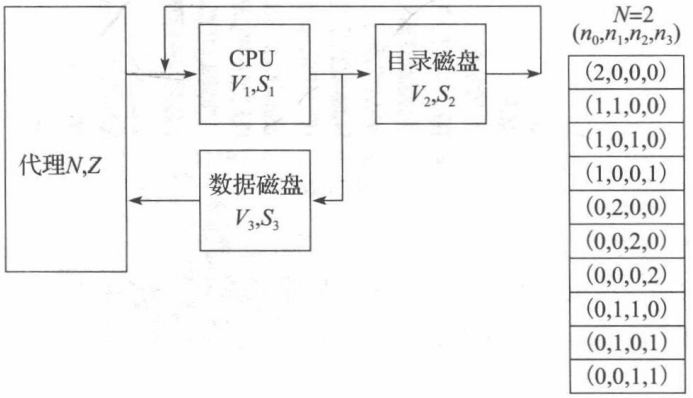


图 9.10 该模型把机票预定系统表示成 4 个服务器：代理、CPU、目录、数据。当 $N = 2$ 时，该系统有 10 个可能的状态，如图右侧所列。每个状态的分量之和 N 必须等于 2。例如，状态 (2，0，0，0) 表示两个用户都在思考，所有服务器都空闲。状态 (0，1，0，1) 表示一个用户的作业正在 CPU 上处理，另一个用户的作业在数据磁盘服务器上运行

189

每个状态的平衡方程表示观察期内进入该状态的转换数等于离开的数量。很不幸的是，状态的数量是随用户数 N 和服务器数 K 呈指数增长的。图 9.10 中的模型在 $N = 2$ 时有 10 个状态；当 $N = 10$ 时，有 2286 个状态；当 $N = 100$ 时，则有 176 851 个状态。对于一个有数千个用户和服务器的网络来说，其平衡方程的数量非常巨大，很难获得求解它们的计算方案。[4]

Jeff Buzen（1973）发现了一种方法，可在 $O(NK)$ 步内从该模型中计算出基本的指标。他的发现是性能分析的一个重要突破。几年之后，Martin Reiser 和 Steve Lavenberg（1980）发现了一个稍微好一些的算法。由于该算法直接计算响应时间、吞吐率、队列长度的平均值，他们的方法被称为*均值分析*（Mean Value Analysis，MVA）。表 9.4 对这一方法进行了总结。

表 9.4 均值方程

方程	释义
（1）$R_i(N) = S_i(1 + Q_i(N-1))$，对所有的 i	当一个作业到达服务器 i，它看到的队列与当系统中的作业少一个时外部观察者看到的基本相同。其响应时间是它到达后队列中的每个作业都被服务一次的时间

（续）

方程	释义
（2）$R(N)=\sum_{i=1}^{K} V_i R_i(N)$	对服务器 i 的每次访问会积累一个本地响应时间
（3）$X(N)=\dfrac{N}{R(N)+Z}$	将利特尔法则应用到循环执行“思考－等待”周期的总时间
（4）$Q_i(N)=X(N)V_iR_i(N)$，对所有的 i	将利特尔法则应用到每个服务器

这些方程给出了服务器的响应时间 $R_i(N)$、系统响应时间 $R(N)$、系统吞吐率 $X(N)$、服务器队列长度 $Q_i(N)$。方程巧妙地通过之前负载为 N–1 时计算的队列长度，构造了负载为 N 时的各个量。均值算法在 $N=1,2,3,\cdots$ 时依次计算四个方程，直到所需的 N 值为止。

经过观察，你很快就能发现表 9.4 中的方程 2 只是一个根据服务器响应时间计算系统响应时间的运算法则。方程 3 是响应时间法则，而方程 4 是利特尔法则 $Q_i=R_iX_i$ 与限制流量法则 $X_i=V_iX$ 相结合。那么方程 1 呢？

方程 1 不是一个法则，而是基于以下简单想法的一个近似估算。当一个作业到达服务器时，它会加入一个长度为 k 的队列，队列长度增加为 $k+1$。队列中的每一个作业，包括它自身，需要平均 S 秒的服务时间。因此，响应时间是 $R=S(k+1)$。k 应该取什么值？ 190

Reiser 和 Lavenberg 从到达定理（arrival theorem）中得到了答案。该定理认为当一个作业到达时，服务器的队列长度与当系统中少一个作业（该作业本身）时外部观察者所看到的一样。换句话说，到达服务器的作业充当了外部观察者。因此，k 的期望值就是 $Q(N-1)$。

对于大的 N 值，该算法会造成浪费。在航空公司的例子中，如果想要得到 $N=1000$ 时的吞吐率和响应时间，你需要先计算 $N=1$，2，…，999 的所有值，然后再把它们丢弃。Yan Bard（1979）与 Paul Schweitzer 磋商，找到了一条捷径。他们为平均队列长度引入了一个近似值：

$$Q_i(N-1)=Q_i(N)\frac{N-1}{N}$$

这种近似根据负载比例，将负载为 N 时的队列长度简单扩展得到负载为 $N-1$ 时的队列长度。将其替代到第一个方程中，得到的方程仅表示负载 N 时的平均值。由此得出表 9.5 中的简化方程。我们可以通过使用这些方程，从猜测队列长度是 N/K（不正确的）开始，对各个量的平均值生成一系列的猜测值，从而完成对方程的求解。在相对较少次的迭代之后，该过程会迅速收敛到非常接近于完全的均值方程所计算出的值。

这一过程可以很容易用电子表格来实现。在选择了两个缺失的参数之后，我们将简化的模型应用到图 9.10，用来回答本章开头提出的两个预测问题（见图 9.11）。

表 9.5 简化的均值方程

方程	释义
$R_i = S_i\left(1 + Q_i\frac{N-1}{N}\right)$，对所有 i	表 9.4 中的方程 1，将其中的 $Q_i(N-1)$ 替换成其近似值
$R = \sum_{i=1}^{K} V_i R_i$	与表 9.4 中的方程 2 相同
$X = \frac{N}{R+N}$	与表 9.4 中的方程 3 相同
$Q_i = XV_iR_i$，对所有 i	与表 9.4 中的方程 4 相同

191

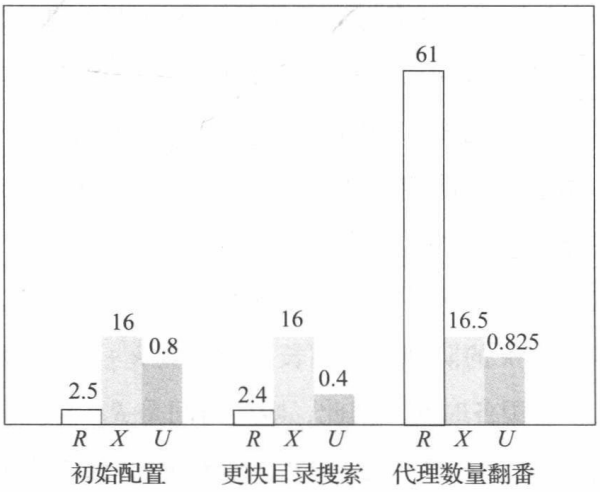


图 9.11 这里的柱状图给出了图 9.10 中建模的航空公司机票预定系统的两个预测问题的模型结果。我们用如下的办法选择 S_1 和 S_3 这两个缺失的参数：选取总的 CPU 时间为 50 毫秒，求解 $S_1V_1 = 50$ 毫秒，得出 $S_1 = 4.5$ 毫秒；选取 S_3 为 60 毫秒。对三个服务器的总需求为 $V_1S_1 = V_2S_2 = 50$ 毫秒，$V_3S_3 = 60$ 毫秒，使得数据磁盘成为瓶颈。根据这些值，利用表 9.5 中的方程就可以得到图中左侧所示的结果。再来考虑这两个预测问题。第一个预测问题是：如果有一种新的目录磁盘结构，可以使得对它的访问次数降低到 5 次，而其他的所有事情维持原样，这时会发生什么？模型给出的结果如图中间部分所示：目录磁盘的利用率降低一半，而吞吐率和响应时间几乎没有受到影响。原因是加速目录磁盘并不会改变数据磁盘是瓶颈的事实。第二个预测问题是：如果代理的数量翻番，增加到 2000 个，又会有什么变化？如图右侧所示，这一改变对吞吐率几乎没有影响，因为 CPU 和目录磁盘已经接近饱和，但这会对响应时间造成显著的负面影响

192

结论

关于计算的一个最难的问题是：它到底需要花多长时间？这个问题在作业相互竞争且形成队列的服务器网络上很难解答。虽然作业和网络都是确定的，但响应时间是随机的。这种不确定性是由网络中作业到达的随机性和服务器上作业服务时间长度的随机性造成的。排队论已被证明是克服这种不确定性并预测计算机网络系统的吞吐率、响应时间及拥塞的一种非常精确的方法（Buzen 2011）。

计算机科学家从20世纪60年代开始使用排队论，他们做出了两项重要的贡献：发展了这一理论并扩展了它的应用。他们发现了通过排队模型计算性能指标的快速算法，而模型公式本身的计算是指数难的。利用这些算法，模型可以被快速地估算。这导致了大量相关的实验研究。这些研究表明，利用简单的模型来预测吞吐率，与测量值相比其误差在5%以内，而响应时间的误差则在25%以内。而正当他们无法解释为什么即使在系统未满足关键模型假设条件下，结果也很好时，计算机科学家又发现了可应用于许多真实系统的更简单的假设，可以得到相同的计算公式。

排队分析的一个重要原则是为系统每个状态的出入流找到平衡方程。平衡方程的解是每个状态所占用时间的比例。这个时间比例就可以用于计算吞吐率和响应时间的公式中。

计算机科学家贡献的最重要的定理之一就是：一个刚到达服务器的作业看到的队列长度，与系统负载少1个时外部观察者所看到的相同。换句话说，刚到达的作业充当了系统的外部观察者（此时系统中不包括刚到达的作业）。这一定理导致了均值方程，从而得到了计算任意计算机网络系统的平均吞吐率、响应时间及队列长度的快速算法。 193

第 10 章

Great Principles of Computing

设　计

简单之中蕴含着复杂。

——Peter G. Neumann

描述一个软件实体时，如果对其复杂性进行抽象，那么往往意味着最本质的东西已经不复存在。好的决策来自于经验，而经验则萌芽于坏的决策。

——Frederick Brooks

我能非常清晰地回想起那样一个时刻：我突然意识到余下生命的大部分时间将会用于发现自己编写的程序中的错误。

——Maurice Wilkes

我们正在探索两种未知事物之间的某种和谐关系：一是还没有设计出的某种形式；二是无法正确描述的上下文。

——Christopher Alexander

1944 年夏末，一群伟大的设计者聚集在一起讨论一种基于存储程序的通用电子计算机器的设计结构。他们是 J. Presper Eckert 和 John Mauchly（ENIAC 项目的首席工程师）、John von Neumann（一位著名的数学家）、Arthur Burks 和 Herman Goldstine（数学家）。他们对先前的计算机器设计项目进行了深刻的反思和讨论。很快，他们明确了一些已经存在的有效设计思想，同时又提出了一些新的思想，这些思想成为后来大多数的存储程序电子计算机的基本设计原理（Goldstine 1993，Wilkes 1995）。通过详细的分析，他们意识到，与现有的计算机器相比，存储程序将会使得计算机的运算速度和效率得到极大的提升。他们实现了一台名为 EDVAC 的计算机器，这台机器于 1951 年开始正式运行。英国剑桥大学的 Maurice Wilkes 使用这些基本原理在实验室中实现了另一台名为 EDSAC 的计算机器，这台机器在 1949 年就开始了正式运行。存储程序计算机已经成为一种得到有效验证的伟大设计（见图 10.1）。

随着程序设计经验的不断积累，这些项目的设计者对计算机做了改进，使得它们更加高效且更少出错。他们发明了索引寄存器实现对列表数据（即现代所说的数组）的访问。他们设计了可用于子程序调用和返回的指令。他们设计了中断系统，使得处理器可

以通过跳转到相应的子程序来应对外部发生的紧急信号。他们还发明了虚拟内存，使得数据和程序可以在内存和外存之间进行自动化的迁移，避免了手工管理操作中可能存在的错误。

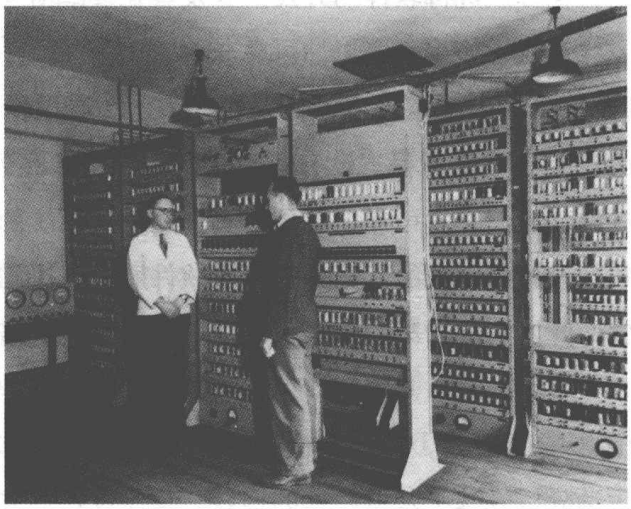


图 10.1　1949 年左右，Maurice Wilkes 和 William Renwick 站在 EDSAC 计算机的旁边。EDSAC 是第一台可实际运行的存储程序计算机，其中使用到了由 Eckert、Mauchly、von Neumann、Burks、Goldstine 等人在 1944 年提出的 6 种设计思想。这 6 种思想分别是：（1）所有部件都应电子化（即除了输入和输出设备之外，不存在机械式运动部件）；（2）数值采用二进制存储（以提高电路的容错能力）；（3）指令集是用户和机器交互的接口（用户通过编写程序来控制机器）；（4）指令是顺序执行的；（5）内部存储器不应该对程序和数据进行区别对待（EDSAC 使用水银延迟线作为其内部存储器）；（6）指令在运行时可以被修改。采用这些基本原理的计算机，其结构通常称为“冯·诺伊曼结构”，因为这些基本原理的第一个发表版本是冯·诺伊曼（von Neumann）基于群体讨论形成的笔记（von Neumann 1945）(图片来源：剑桥计算机实验室历史遗迹发掘项目。该图片得到授权)

这些机器孕育了一种全新的专业领域——程序设计，并且最终促成了软件产业的形成[1]。第一代的科学领域程序员设计数值计算方法。第一代的商业领域程序员则设计处理大规模数据的计算方法。通过实践，这些程序员意识到程序设计是一种困难且非常容易出错的活动。系统设计者因此开始关注如何使程序设计更加容易和可靠。其中一个重要进展是高级程序设计语言的发明：Fortran（1957）、Lisp（1958）、Algol（1958）、Cobol（1959）。这些高级语言可以让程序员通过非常简洁的方式表达出复杂的算法。编译器将这些源程序自动翻译为相应的机器代码。调试工具帮助程序员发现并修正程序中的错误。经过仔细测试和验证的通用程序库得到了广泛的使用，典型实例包括数学软件库和系统功能软件库等。

除了对工具、方法、程序库的关注外，一个得到广泛认同的观点是：大多数的软件都

不可靠、不可信。1968年，一群重要的软件专家聚集在著名的NATO软件系统研讨会上。他们宣称整个软件产业将会处于持续的危机之中，因为软件规模和复杂度的不断增加，远远超过人们掌握的软件开发工具和技术的能力范围。他们呼吁成立一个名为“软件工程”的领域，通过严格的工程化方法进行软件的开发。开发者在实践活动中发明了很多新的工具和方法来减少程序中的错误，使得程序更加可靠。

196
~
197

18年之后，Fred Brooks，一个软件专家同时也是IBM360操作系统开发项目的前主管，提出了一个著名的论断：软件工程“没有银弹”（Brooks 1986）。他指出，虽然软件开发工具有了长足的进展，人们开发可靠、可用、安全的软件系统的能力并没有在实质上得到提升。他认为，软件设计最困难的部分是获得对软件所要解决问题的清晰理解。这不是一件容易的事情。软件开发的成功与否在很大程度上依赖于能否培养出具有合适技能的人才。

这是一个具有深远影响的结论。在很大程度上，程序设计的成功依赖于工程师的技能，而不是形式化的数学分析或基于基本原理的推导。程序设计的成功还依赖于设计者的历史经验积累；这些经验告诉设计者哪些方案是可行的，而哪些又是不可行的。

目前来看，设计者在计算中承担了核心的角色。设计者使用他们所掌握的技能对软件和硬件的形态产生了重要影响：他们创造了能够按照预期运行的计算过程，对相关领域的实践活动提供了更加高效的支撑。在设计者眼中，硬件和软件仅仅是一种工具。本章主要关注设计者如何克服复杂性设计出符合预期的软件系统。

计算机科学的重要挑战之一是如何设计和构造大规模的计算系统，满足用户的可靠、可用、安全等指标（Dependable、Reliable、Usable、Safe、Secure，简称为DRUSS）。最有效的方法已经被集成在程序设计语言和操作系统的设计之中，使得大范围的人可以从中获益。本章不仅仅关注计算的设计原则和技巧，还关注设计者为了实现DRUSS的目标而使用的一些结构。

什么是设计

设计是一个人们非常熟悉的概念，出现在时尚、产品、建筑、工程、科学、软件开发等众多领域中。设计是一种通过创造和规范特定的制品来解决问题的过程。例如，在软件开发中，设计意味着精心制作出满足用户预期的软件。软件设计者想让软件对实践产生帮助作用，因此总是不断地寻找软件的潜在用户。软件设计者已经积累形成了很多的实践型智慧，表现为一组软件设计的基本原则（如，关注点分离、模块化、抽象、分层、整体性、功能性、弹性、美感、持久性等）。设计原则指导着我们去构造出对用户群体有用和

有意义的计算系统（Norman 2013，Winograd 1996）。

需要注意的是，设计原则和模式并不是设计技能的全部。设计是一种非常精细和深 198
奥的技能。它所涉及的很多方面需要在一个经验丰富的导师的带领下才能逐渐领会。一般而言，设计师的工作需要去深入聆听用户群体的关注点、问题和兴趣，然后再寻求通过对已有系统和技术的组合来应对这些关注点、问题和兴趣。设计者还需要观察用户对设计方案的反馈，进而提出更好的设计方案。设计工作会涉及多轮的“评估－学习－改进”的循环。同时，设计也具有非常厚重的历史感，因为设计师要学会与那些持续变化的已有系统和关注点和谐共存。

在软件设计师出现以前，设计在很多领域中就已经受到了关注。建筑设计师致力于让其建造的建筑和桥梁可用、安全、美观、持久耐用[2]。服装设计师致力于让衣服更加时尚和舒适。工业设计师致力于生产出满足 Rams 原则的消费产品——创新、有用、美观、易理解、沉稳、忠实、持久、细致、环保、简单[3]。工程师致力于建造复杂且安全可靠的工程系统。“设计思维”这个词就是用来描述在满足上述要求的情况下解决问题的思维方式（Denning 2013）。

软件设计师需要满足两种类型的标准：以 DRUSS 目标为代表的传统工程设计标准和以 Rams 原则为代表的工业化设计标准。在计算领域内，已经涌现出了不同的设计流派，每一种流派都或多或少地关注了这些设计标准的不同子集。

经验丰富的设计师通常会采用原型的方式去观察其设计制品的工作效果以及用户对这些制品的反应。1967 年，Maurice Wilkes（1913—2010）在其图灵奖获奖演说中强调了这样一点：在计算机的早期发展阶段，研究者总是非常愿意去建造实验性的计算机，且很少关注这些计算机是否会得到真正的商业应用。这些实验工作产生了系统性的知识体系，告诉人们哪些东西可用，而哪些东西又不可用（Wilkes 1968b）。在 1995 年的回忆录中，Wilkes 强烈批评了当时存在的一种现象，即：忽视历史，总是从头进行设计。没有历史形成的知识体系，设计者总是会重复犯下相同的错误（Wilkes 1995，p.90）。今天，我们可以看到，个人计算机操作系统的设计者似乎就走过了相同的历程，经过长期的困扰之后才发现原来解决方案在很多年前就已经存在了。和 Fred Brooks 一样，Wilkes 相信好的设计是一组多维度的技能，需要精心的培育才能形成。 199

许多设计者只是关注如何对已有的实践活动进行自动化，而不是去创造新的实践活动。发明于 1979 年的自动化电子表格就是这样的一个例子：它模仿了标准商业实践中数字的显示和处理活动，并且通过自动化，极大提高了运算的速度（见图 10.2）。出现于 1971 年的 ATM 机则是另外一个例子：它实现了对银行存取款服务的自动化支持。

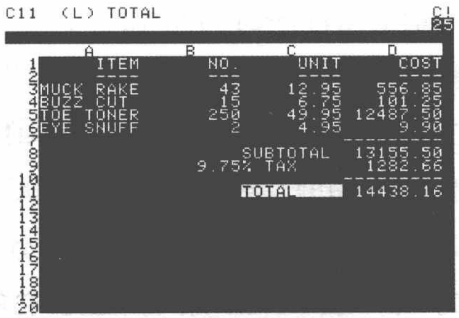


图 10.2 VisiCalc，第一个电子表格程序，出现在 1979 年 Apple II 个人计算机中。它的设计者，Bob Frankston 和 Dan Bricklin，在这个软件中实现了对标准会计活动的自动化支持。数字可显示在行或列中。任何地方都可以做上标记。标准的运算操作以公式的方式显示在对应的单元格中，在屏幕显示的则是公式的运算结果。商业用户发现电子表格非常易于使用，而且能极大地提高他们的工作进度。VisiCalc 展示了一个非常重要的设计原则：尽可能保持设计与用户实践活动的一致性

软件系统的准则

第 5 章讨论了程序中存在错误这一问题：程序员在程序设计过程中总是会不可避免地引入错误。有很多可能的原因会导致错误的引入，例如：设计者的设计意图出现了错误，或者用户对最终产品的期望发生了错误。错误是程序员无法避免的一个重要挑战，也促使了工程化设计过程在软件开发中的广泛使用。即便如此，彻底消除程序的错误仍然是一个非常困难的事情。

200

虽然不是一种完美的解决方案，工程化的设计过程仍然被大多数的软件开发活动所使用。为了尽可能取得成功，设计者需要遵循 5 种重要的准则：

1）需求：软件是否具有足够明确的需求？

2）正确性：软件是否能够正常地工作？

3）容错性：软件在各种条件下是否都能持续工作？

4）时效性：软件是否在有效的时间内完成了工作？

5）适用性：软件是否能够与环境和谐交互？

为了实现这些准则，设计者已经将相关的设计原理融入到程序设计语言、工具、操作系统以及网络中。

需求

设计者知道软件所要完成的工作是什么，并且能够通过规约的方式准确地表达出这些需求。这一目标说起来容易做起来难。表达需求绝对是一个挑战，因为与用户的交流并不总是能够发现他们的真正需求。很多设计者因此转而只关注技术，而不再愿意聆听用户的

声音（Norman 2010）。让软件的目标用户与软件原型发生交互，是一种更加可靠的方式去发现用户在表述其需求时可能存在的错误，或去明确软件应该具有的最有价值的特性。

正确性

源程序或机器代码程序完全准确地表现出规约中说明的行为。正确性很难完全实现，因为很多时候需求本身就是模糊的。想通过证明的方式来确认正确性通常不具有可行性。试验性方法很多时候是唯一的选择。

关于计算正确性的梦想可以追溯到 Charles Babbage（1791—1871）。他非常关注数学计算表中（如，三角函数计算表、指数计算表等）存在的错误。这些数学计算表是通过*差分方法*产生的——每一行上的数据都是在先前数据的基础上增加一个小的*偏差*而产生的。这样的计算过程完全是通过人力的方式进行的。一行中的一个微小错误都有可能被不断地传播和放大，从而使得最终的计算结果产生巨大的偏差。Babbage 向人们展示，如果在导 201
航计算表中出现了这样的错误，就有可能导致船只的沉没。1823 年，Babbage 说服英国政府资助其构造一台分析机：这台机器可以非常快速地计算出导航计算表，而且绝对不会出现人力计算中可能引入的错误。遗憾的是，Babbage 在分析机的构造上并没有取得太多的进展。英国政府最终在 1842 年放弃了对分析机项目的资助。瑞典的两位工程师 Georg 和 Edvard Scheutz 在 1843 年复制了 2 台差分机，但是，这两台机器的性能非常不稳定且需要进行繁琐的设置，因此很少有人愿意使用它们。

这一段历史插曲告诉我们计算机器从来都无法脱离错误：它们还使得错误的原因更加多样化。设计者必须去证明机器代码准确实现了预期的功能。如果机器只是近似实现了预期的功能（像差分机那样），设计者还需要进行额外的分析和证明，以确保这种近似产生的错误不会对运算结果的价值带来实质性的影响。

计算的设计者对自动正确性验证这一可能性非常感兴趣，即：机器能够自己构造出正确性的证明过程。自动验证的最新进展是模型检查（Clark & Emerson 1981，Quielle & Sifakis 1982，Clark 2008），其基本思想是将软件系统建模为一个有限状态机，然后验证这一有限状态机满足一组时序逻辑公式。有限状态机模型表达了实际机器所能观察到的行为以及这些行为之间的迁移关系。时序逻辑公式表达了软件系统行为的形式化规约。模型检查技术已经成功地应用于具有大规模状态的实时系统的验证[4]。

容错性

软件及其宿主系统能在出现微小错误的情况下持续运行，能在出现较大错误的情况

下停止工作。支持容错的一种方式是冗余，即：将硬件和数据复制为多个副本，以保证在一个副本失效的情况下，其余的副本仍能进行正常的工作。支持容错的另一种方式是错误限制，即：通过对进程的操作环境进行结构化以确保进程不会访问与其计算无关的资源，限制或消除系统的超级用户模式[5]。

一个相关的原则是最小特权原则，即：系统的设计者应确保每个进程在默认状态只会访问与其计算相关的一组最小资源。最小特权原则可以用于实现错误限制。

错误限制的机制可以分为三种：

202 1）静态检查。例如，由编译器进行的各种类型检查：浮点操作仅作用于浮点数、将字符串传递给子程序时确保子程序的参数接收字符串数据、文件操作句柄仅传递给文件系统操作函数等。

2）动态检查。一个典型实例是数组访问越界检查，即：使用一个索引值访问数组中的元素时确保索引值在合法的范围之内。另一个例子是缓冲区溢出保护，即：一个过程的参数不应该超过为该参数分配的存储空间。编译器可以通过在机器代码中插入相应的代码片段来保证动态检查的自动实施。

3）对软件运行的宿主环境进行动态检查。例如，操作系统会强制检查传递给文件管理系统的参数确实指向了某个文件。另一个例子是，操作系统会强制进行存储空间分区，确保一个进程中的指令只会对本进程所拥有的存储区域中的数据产生影响。

静态检查的实施成本最低，但无法应对错误输入、数据污染、设备失效等动态错误。动态检查应尽可能地避免，因为频繁的内部检查会导致系统开销的增加——可以想象，如果在一个持续执行的循环中进行数组访问越界检查，将会对循环的执行效率产生多大的影响。因此，我们需要使用环境信息对动态检查进行优化。后文会给出相应的例子。

容错的另外一种重要手段是“端到端检查”，其被广泛使用在分布式系统和网络中（Saltzer et al. 1984）。端到端检查的基本思想是只在通信的两端进行错误检查，而不对通信的过程进行错误检查。如果在接收端发现错误，则要求发送端重新发送该数据。这一手段是互联网 TCP 协议的核心（Tanenbaum 1980，Comer 2013）。

时效性

系统必须在预期的时间区间内完成任务。典型的支持技术包括算法分析、排队网络分析、实时系统时限分析等。

适用性

系统能够与其使用环境和谐交互。适用性的挑战在于 DRUSS 目标集的上下文敏感

性，而很多上下文因素即使对于一个有经验的设计者而言也无法轻易被发现。

前文提到的 ATM 自动取款机是机器与其使用环境之间和谐交互的一个代表性实例。有经验的软件设计者会意识到他们所设计的不仅仅是某种机制，而更是一种有效激发用户 203
参与的新型实践活动。软件与用户熟知的实践活动越接近，用户就会越容易接纳软件。

其他具有良好适用性的软件包括软件游戏、电子表格、亚马逊购物网站、贝叶斯垃圾邮件过滤器、语义网络、谷歌搜索引擎等。具有较差适用性的软件包括操作系统中未被识别的错误代码、商业电话服务中的自动语音菜单、在线技术支持与帮助系统等。

设计原理、模式和示意

设计者在实践中积累了很多有用的经验来应对上述挑战。这些经验可以表述为设计原则、设计模式和设计示意三种形式[6]。

设计原理是对设计决策中所需的技术和策略的描述。设计原理的目标是让设计能够产生满足上述 5 种准则的设计方案。

设计模式是对设计者经常遇见的一些共性场景的描述。设计模式通常会告诉程序员如何去组织程序，或如何去实施程序设计过程，以达到最好的设计效果。

设计示意通常包括一些更宽泛的设计经验和建议等，不过不像设计原理和设计模式那么形式化、正规化。

下面三个小节依次给出设计原理、设计模式、设计示意的若干实例。需要注意的是，在实践中我们发现了数量非常多的原理、模式和示意。在下面的实例中列出了 115 条。但是，这并不意味着程序设计这个领域的不成熟性，而只是说明一个好的设计者通常需要掌握不同级别的技能。

原理

设计原理的一个典型实例是由 Jerome Saltzer 和 Michael Schroeder 在 1975 年提出的一组关于信息保护的设计原理（见表 10.1）。2009 年，Saltzer 和他的同事 Frans Kaashoek 对近 30 年以来的实践经验进行了系统性的总结，形成了 25 条经过广泛验证的系统设计原理以及相关的附属性原理。由于篇幅关系，我们不在此对其进行更详细的介绍。

表 10.1　Saltzer 和 Schroeder 提出的关于信息保护的设计原理

原理	描述
机制的经济性	保持设计的简单与细粒度
故障时的安全默认值	拒绝默认访问；只有得到显式授权才允许访问
完全中立	对每一个对象的每一次访问都进行检查

（续）

原理	描述
开放设计	不要假设攻击者不了解系统的设计
权限分离	对访问的控制检查要基于多种信息源
最小化权限	对任何一个进程，只赋予与其任务执行相关的权限集合
最小公用机制	不允许一个进程修改共享的状态信息，以避免共享信息遭到破坏
心理可接受	保护机制应易于使用，或至少很容易可以将其屏蔽

模式

计算领域的研究者从不同的角度探索了实现系统设计5种准则的多种可能的方式，
204 形成了不同的思想流派。一些流派推崇“过程模型”，例如：瀑布模型、螺旋模型等。另一些流派则推崇“设计途径”，例如：提高设计活动的参与性、以用户为中心、敏捷、模式等。不同思想流派的最终目标是一样的。他们之间的差异主要体现在对不同准则的相对重要程度存在不同的理解。Barry Boehm（2002）认为，标准的工程化设计方法提倡严格甚至刻板的设计过程，代表了一种极为严谨的设计思想；敏捷方法则与之相反，代表了一种极为灵活的设计思想。他认为，严格的设计过程适用于对可靠性和安全性有较高要求的场景，敏捷方法则更适用于对可用性和演化性有较高要求的场景。他呼吁，这两种不同的设计思想需要相互协作、相互融合，以形成关于系统设计的更优模式。

20世纪90年代初，敏捷思想流派的一组程序员，受到了建筑学家Chrisopher Alexander（1979）关于建筑设计模式的启发，发起了“软件模式共同体”运动。一个软件模式刻画了程序员经常遇到的一类场景，并且给出了应对这种场景的最佳程序设计方案。他们的早期工作成果之一是一本关于软件模式的手册（Coplien和Schmidt 1995）。从那时起，这个共同体不断提出了更多的设计模型。维基百科的设计模式词条中一共汇集了4类58种设计模式[7]：

第1类：创建型模式（对对象和接口的创建进行管理的模式）。例如：对不再使用的
205 对象进行回收，以避免资源的浪费。

第2类：结构型模式（对代码结构进行管理的模式）。例如：将一组相关的原则组织为一个单独的概念实体。

第3类：行为型模式（对模块的行为进行管理的模式）。例如：通过为变量默认赋值为一个空对象，以避免对变量的空引用。

第4类：并发型模式（对对象的并发访问进行管理的模式）。例如：当一个对象需要被互斥访问时，为该对象关联一个访问监视对象。

软件模式共同体信奉经验至上的理念：他们总是从实践中不断地进行学习，总结提炼出相应的设计模型，并通过实践对这些模式进行测试和改进。

示意

Butler Lampson，一位杰出的软件设计者，在1983年总结了一组设计指导原则，并将这些指导原则命名为“设计示意”，因为其中的每一种指导原则都不具有普遍适用性，它们是优秀设计者随时间发展的一种方向辨别力。表10.2以标语的方式简要表达了这些设计示意。我们在此不对其作进一步解释。需要注意的是，设计不仅仅是一种技术活动，其中很多方面具有很强的艺术性。Lampson对软件设计的最佳实践也进行了很好的总结。

表10.2　Lampson提出的设计示意

	正确性／适用性	速度	容错
用例	将正常情况与异常情况分离	安全第一 降低负载 端到端	端到端
接口	保持简洁性 做好一件事 不要一般化 保持正确性 不要隐式化 使用过程参数 让使用者决策 保持接口稳定	尽可能快 资源分离 静态分析 动态变换	端到端 日志更新 操作原子化
实现	随时舍弃 保守秘密 复用好的思想 分而治之	缓存计算结果 使用设计技巧 使用暴力破解 在后台计算 批处理	操作原子化 使用设计技巧

206

软件系统的设计原理

如上文所述，软件工程文献中已经记录了大量的被实践所验证的软件系统设计原理。其中，最核心或最基础性的设计原理已经被落实到编程语言、应用程序以及操作系统的结构之中。目前，最为流行的结构包括：

模块化　　类层次
接口　　分层
抽象　　虚拟机
信息隐藏　　复用
封装　　对象和包
分解　　版本控制
分离式编译　　客户端－服务器
功能级别

这些结构可以作为工具帮助设计者应对一些重复出现的模式。但是，如果设计者对这些结构所针对的模式缺乏清晰的了解，那么，就有可能对这些结构作出不恰当的使用。下面，我们简要介绍涉及所有上述结构的6种模式：层级式聚合，封装、级别、虚拟机、对象、客户端－服务器。

层级式聚合

层级式聚合指的是一个包含一组相互交互的子对象的对象（即可识别的软件或硬件成份）是一个更大粒度对象的构成成分。你可以直接与一个对象交互，而不必关系其内部细节。如果进入一个对象的内部，你可以看到一组更小粒度的对象及其之间的交互关系。这样，不同粒度的对象通过逐层聚合的方式形成一个层级结构。这个层级中的每一个对象都对上层对象和下层对象进行了很好的隔离。

层级式聚合在自然界中普遍存在。物理对象按照其尺寸，自然形成了一个具有45层的层级式聚合结构：夸克、电子、质子、原子、分子在10^{-18}米的尺寸级别上，处于这种层级结构的底层；行星、恒星、星系、星云、类星体在10^{26}米的尺寸级别上，处于这种层级结构的顶层。

在生物学中也存在类似的层级式聚合结构，其涉及的层级包括DNA、基因、细胞、
207 器官、神经系统、植物／动物、社会系统。在数学中，分形也是一种层级式聚合结构，其特点在于部分与整体具有相同的结构。在计算领域中，局部性原理即是层级式聚合结构所展现的一种性质：相邻的两个成分之间发生交互的频率远大于相聚较远的两个成分之间交互的频率；与一个对象的一次交互，会触发对象内部成分之间的一个较长的交互序列。

在上文列出的设计结构中，模块化、抽象、信息隐藏、分解等结构与层级式聚合具有较强的相关性。

模块化指的是将一个大的系统分解为一组更小粒度的模块，模块之间通过清晰定义的接口进行交互。

抽象指的是为一个事物定义一个简化版本以及作用在该简化版本上的一组操作。例如，比特（0或1）是对任何能够存储二值状态信息的媒介的一种抽象，计算的过程涉及对比特的读和写两种操作。抽象是人脑具有的最基础的能力之一。通过去除细节、保留更本质的东西，抽象提供了一组适用于所有情况的简单操作。一个抽象体即对应于层级式聚合结构的一个聚合体，形成层级结构的过程即是抽象的过程。在传统科学中，抽象主要用于解释：抽象定义了基本规律，描述了事物的运作过程。在计算领域中，抽象的作用更大：不仅定义了各种计算对象，还可以实施各种操作。

文件（一个具有名字的比特序列）是对所包含的数字对象的一种抽象，这些数字对象包括文本、图形、电子表格、图像、视频、声音、目录等。文件系统则提供了可以作用于任何文件的一组操作，包括：创建、删除、打开、关闭、读、写等。任何一个程序的输出（表示为比特序列）都可以被存放在一个文件中。文件系统并不关心特定类型文件在内容格式上的差异性，而只关心如何存储和读取比特数据。

信息隐藏指的是如何对用户屏蔽实现过程中的细节（Parnas 1972）。通过信息隐藏，用户无法获知实现细节，可以避免用户对实现细节可能产生的不必要的依赖，从而避免由于实现细节的改变而可能产生的错误。在层级式聚合结构中，对一个聚合体的内部结构进行隐藏体现了一种设计决策，其结果是具体实现方式的改变不会导致聚合体外部可见行为的改变。一个软件模块通过对外提供的简单接口隐藏了内部的实现细节。

文件系统很好地展示了信息隐藏的设计原理。用户仅看到文件，而不用关心背后的复杂实现细节，例如：磁盘、磁盘驱动器、磁盘地址、记录、索引表、缓冲区、文件控制块、文件 208
在内存中的副本等。信息隐藏给用户带来了两个方面的利益。第一，用户只需要关心对文件的打开、关闭、读、写等操作，而根本无需关心文件被存放在哪个磁盘上、文件如何被分解为更细小的存储单元，以及如何对文件缓冲区进行管理等底层细节。第二，软件工程师可以在不对用户造成任何干扰的情况下对文件系统的内部实现细节进行持续的优化和改进。

实践中也存在不具有信息隐藏的抽象。例如，一个组织中的层级结构是按照角色对人员进行的一种抽象。但高层决策者可以进行微管理，即跨越层级对下层的人员进行管理。

分解指的是把一个大的问题分解为更小粒度的成分；每一个成分可以进行独立的设计，并通过组装的方式形成一个完整的系统。在一个层级式聚合结构中，识别出聚合体的内部成分即是一种分解。模块即是对其内部成分的一种抽象。

对于一个大型系统而言，通过设计将其分解为模块和接口，往往还不够。当这些构成系统的模块被开发完成并组装在一起后，虽然每个模块都正确地实现了各自的规约，整个系统仍然可能无法正常工作。这其中的问题在于，除了模块的内部实现之外，构成系统的另外一个重要成分是模块之间的交互；每个模块的设计者无法对模块之间的交互进行测试。因此，对系统的整体进行测试是系统设计中不可缺少的一个环节；通过这个环节，可能会发现单个模块设计上的缺陷，从而对其重新设计，直至整个系统能够正常工作。

封装

将软件封装为最小特权域是封装这种设计原理的一个典型实例（Dennis 和 Van Horn 1966，Fabry 1974，Saltzer 和 Schroeder 1975）。最小特权域对于运行不可信的软件尤其

有效。一个进程的保护域表现为一个能力列表，即：一个指向该进程被授权访问的所有对象的指针列表。能力列表的详细内容参见第 7 章。一个进程只能访问出现在能力列表中的对象；对其他对象，由于无法获知其地址，则根本无法访问。

一般而言，一个进程在其声明周期中只会使用一个能力列表。但是，当一个进程需要使用一个不可信的过程调用，该进程可以切换至一个较小的保护域上；这个域中仅包含
209 该过程调用完成正常工作所必需的一个最小能力列表。因此，被调用的过程无法访问这个最小能力列表之外的对象。当该过程调用结束后，进程又恢复至原来的保护域。无论被调用过程内部是否包含错误或恶意代码，这个过程也无法对最小能力列表之外的对象造成任何的伤害（见图 10.3）。

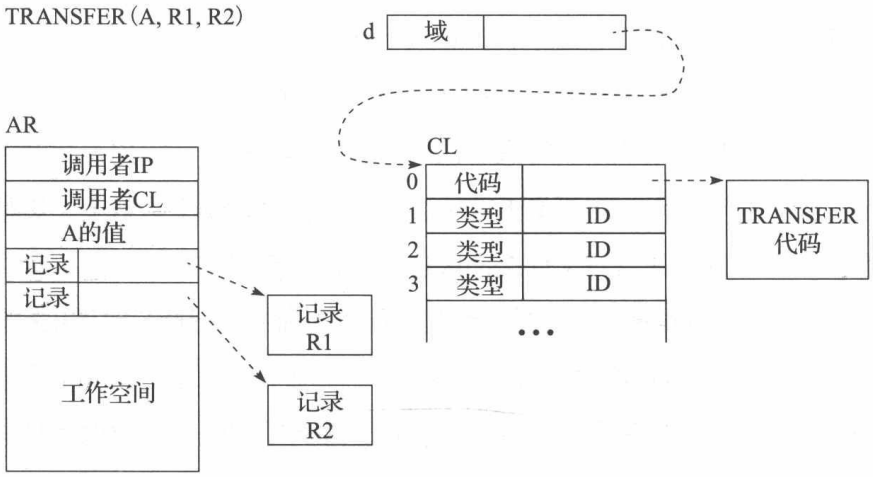


图 10.3 基于能力的操作系统通过特定的结构来防止错误的扩散。这张图给出了对一个不可信的转账程序进行封装的结构。这个转账程序将数额为 A 的资金账户记录 R1 转移到账户记录 R2 中。当这个名称为 TRANSFER 的程序被调用时，它的能力列表被设置为特权域 d。这个特权域中包含 TRANSFER 程序的代码以及被编译器所附加的一组内部对象。对于该程序的一次正常调用会被编译器替换为一条新的指令“ENTER d”（此处，d 即为特权域 d）。ENTER 则会创建一条激活记录（AR），将当前的能力列表设置为 d（通过改变 CPU 中特定寄存器的值），并且将指令指针指向能力列表中编号为 0 的能力的起始位置。然后，TRANSFER 程序就被限制在特权域 d 中进行执行。这时，激活记录（AR）中还会记录下调用者所关联的能力列表，使得在 TRANSFER 程序执行完毕后能力列表能被复原。激活记录（AR）中还会包含两条分别指向 R1 和 R2 的能力。TRANSFER 程序在执行过程中，只能访问到 AR 中所存在的这两条记录，而无法访问到任何其他能力

级别

级别是另一种形式的层级式聚合结构。处于同一级别的所有对象在相互之间的交互以及与高层对象的交互方面具有同等的地位。前文提及的自然界所展现出的级别结构即展

示了这样的性质。在原子级别上，我们主要关注化学键接、缺电子化合物、分子形状等问题，而不太关注原子属于哪一个物品、原子内部的各部分之间如何结合在一起等问题。

级别原理已经被用于对非常复杂但具有可证明正确性的软件系统的结构进行设计。它被首次使用在操作系统的结构设计中。1968 年，Edsger Dijkstra 在 Technische Hogeschool Eindhoven 开发完成了一个被后人称为“ THE operating system”的操作系统。Dijkstra 将这个操作系统分为 7 个级别；每一个级别包含了一组软件构件，实现了一种特定的抽象概念。例如，对于“进程”级别，它在处理器的基础上抽象出一种除非等待信号否则持续向前的计算过程。所有高于“进程”级别的软件构件可以直接在进程的基础上进行程序设计，而不需要关心中央处理器是如何在不同的进程之间进行上下文切换的。“进程”级别较好解决了对单一中央处理进行多路复用的问题。10 年之后，SRI 公司的一个研究小组构造了一个安全性可证明的操作系统：这个系统具有 14 个层级；而且可以证明，对于一个级别，只要其下的所有低层级别都是安全的，那么，这个级别也是安全的（Neumann et al. 1980)(见图 10.4)。

级别	名称	对象
9	Shell	接口语言（负责启动应用程序、管理窗口和用户事件）
8	目录	目录、目录树、路径名
7	流	文件、设备、管道
6	虚拟机	虚拟机
5	线程	同一地址空间的多个线程
4	进程间通信（IPC）	消息、端口、套接字
3	虚拟存储	地址空间、存储页
2	进程	进程、就绪列、信号量
1	底层输入 / 输出	设备、驱动程序
0	硬件级别	中断、过程、调用栈、指令集、逻辑门

图 10.4　图中给出的 10 个级别是通过对多种操作系统的级别结构分析综合后形成的（ Denning et al. 2000)。硬件是最低级别。其上的每一个级别分别实现了面向特定类型对象的一组操作例如对文件的读或写操作。每一个级别实现的操作是通过对低级别操作进行组合而形成的。级别 1 ~ 5 形成了操作系统的微内核，包含必须在管态模式下执行的最小功能集合。微内核往往只会占用非常少的存储空间。级别 5 ~ 9 通常具有分布式的特点，即：一个可信网络中的任何机器都可以访问存在于该网络中的任何一个对象，而无需考虑该对象的物理位置。从用户级别到硬件级别的时间刻度相差 10^{15} 个数量级，这使得操作系统成为最复杂的人造系统之一

互联网工程师已经将通信协议软件组织为不同的层次。层次类似于级别：两者都将软件进行分层，并使得上层功能的开发可以在低层功能的基础之上进行。例如，互联网传输协议 TCP 即建立在若干低层协议之上，具体包括 IP 协议、路由协议、数据链接协议、物理信号协议（ Tanenbaum 1980，Comer 2013)。层次不同于级别的重要一点在于：层次之间的访问是通过数据的向上或向下传输实现的；而一个级别只能访问低层级别，且只能通

过直接过程调用的方式进行访问。

210 ~ 211 无论对于操作系统或网络，级别原理都极大地促进了系统的构造、正确性证明以及测试，其主要原因在于级别原理使得系统的构造可以逐层进行。

虚拟机

虚拟机（Virtual Machine，VM）使用一台计算机来模拟另外一台计算机。这个思想来源于通用图灵所蕴含的模拟原理。虚拟机这个术语有四种具体含义。

第一，虚拟机指的是对任何抽象计算机器的模拟。一个抽象计算机具有一组操作，这些操作可以用于对计算机存储器中的数值进行处理。每一个操作类似于硬件计算机的一个指令。这组操作通常称为应用程序接口（Application Program Interface，API）。抽象计算机的用户只能通过 API 与计算机发生交互，而无法获知 API 内部的具体实现细节。操 212 作系统中管理特定对象的子系统就采用了这种方式进行组织，即通过 API 对特定的对象（例如文件）进行管理。图 10.5 展示了文件系统的基本工作原理。

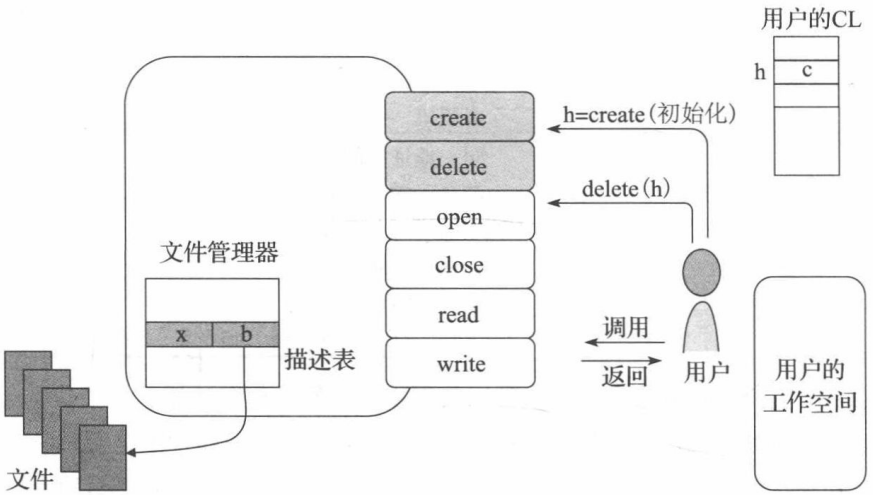


图 10.5 文件管理器由两个部分组成：一个具有 6 种指令 / 操作的虚拟机；一个用于存放文件的存储器。文件管理器的内部结构则被封装起来，用户无法看到文件结构、文件描述表、缓冲区、操作的实现代码等内容。这 6 种操作的工作流程如下所示。创建操作 create 在存储器的位置 b 生成一个新的文件，为这个文件确定一个唯一标识符 x，并且将 x 封装在一个新的能力 c 中。创建操作返回的这个新的能力 c 被放置在用户能力列表中一个方便的位置 h 上。创建操作还在内部文件描述表中记录了关联关系（x，b）。这样，用户通过本地名称 h 就能引用这个文件。删除操作 delete(h) 将文件 h 移除。打开操作 open(h) 在 RAM 缓冲区中复制了文件 h 的一个副本（为了访问上的快速性）。关闭操作 close(h) 将文件 h 在缓冲区的副本写回磁盘并将副本删除。读操作 read(h) 和写操作 write(h) 在缓冲区副本和用户工作空间进行字节传输。这 5 种以 h 为参数的文件系统操作从 h 中抽取出文件的唯一标识符 x，在文件描述表中定位到关联关系（x，b），然后将具体的操作应用至存放在位置 b 的文件

第二，虚拟机指的是对硬件计算机的模拟。虚拟机通过一组子程序来模拟硬件计算机的机器指令。这一思想在实践中的应用始于 20 世纪 50 年代末，当时，第二代计算机开始取代第一代计算机。为了能够运行那些为上一代计算机编写的程序，新一代的计算机在其指令集中包含了一组微代码，用于模拟上一代计算机的指令集。这样，为上一代计算机编写的程序就可以在模拟模式下运行于新一代计算机上。当为上一代计算机编写的程序在新一代计算机上重新编译后，则可以在正常模式下以更快的速度运行。这种模拟模式可以在 Parallels、VMware 以及 Hyper-V 等虚拟机软件中看到，这些虚拟机软件可以模拟出运行有特定操作系统的硬件计算机。几乎在每一种商用操作系统上都存在的 Java 虚拟机（Java Virtual Machine，JVM）则模拟了一种可以执行 Java 字节码的硬件计算机，这使得 Java 程序具有非常优良的可移植性。

第三，虚拟机指的是对一台机器上具有独立存储分区的若干计算机的模拟。这种模拟出现在 IBM VM 370 以及后继的操作系统中。除了内存容量存在差异之外，这些 IBM 虚拟机是对 IBM 大型计算机的完美模拟（见图 10.6）。一个类似的思想是出现在 Mac OS 和 Windows 等操作系统中的多任务特性，它使得虚拟机具有几乎与硬件机器相同的执行速度，且没有显著的性能损失。

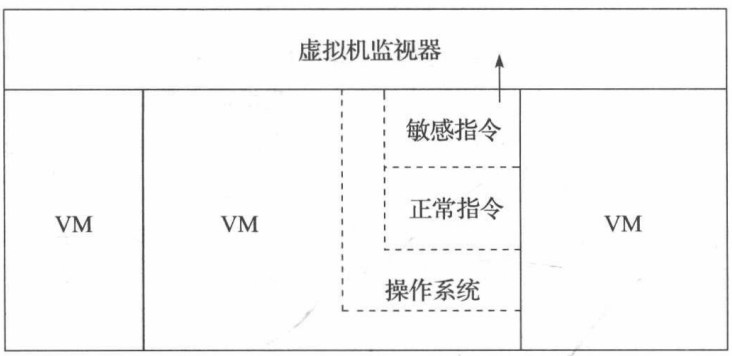


图 10.6　一个虚拟机操作系统将内存划分为若干互不相交的区域，每个区域会分配给一个虚拟机（VM）。一个虚拟机包含一个 CPU 拷贝，虚拟机的存储器中则会包含一个操作系统拷贝。因此，不同的虚拟机可以运行不同的操作系统。虚拟机监视器是一个全局的操作系统，它负责对虚拟机的资源进行分配和管理，保证不同的虚拟机能够和谐相处在同一硬件机器中。为了实现这一目的，CPU 的指令集被划分为两类。正常指令仅作用于所在虚拟机的存储空间，能够被直接执行。敏感指令会对整个系统（包含所有的虚拟机）的状态造成影响，可能不能被直接执行。例如，“增加存储器空间”和“关闭中断”即是两条敏感指令。如果 CPU 尝试执行一条敏感指令，将会触发中断，并使得虚拟机监视器获得控制权。例如，改变存储器容量的敏感指令会被虚拟机监视器截获，并确保这条指令的执行不会覆盖其他虚拟机的存储空间。在大多数情况下，这种类型的虚拟机只会执行正常指令，因此，在执行速度上不会有任何的损失

第四，虚拟机指的是操作系统内程序运行的一种标准环境。这种思想来源于MIT的Multics操作系统（Organick 1972）以及贝尔实验室的UNIX操作系统（Ritchie和Thompson 1974）。这些系统将进程定义为运行在虚拟机上的一段程序。这里，虚拟机指的是一种标准模版，定义了如何支持程序的输入和输出以及如何支持虚拟机与派生出的子虚拟机之间进行交互。每一个用户程序都会被嵌入在一个标准虚拟机中执行（见图10.7）。

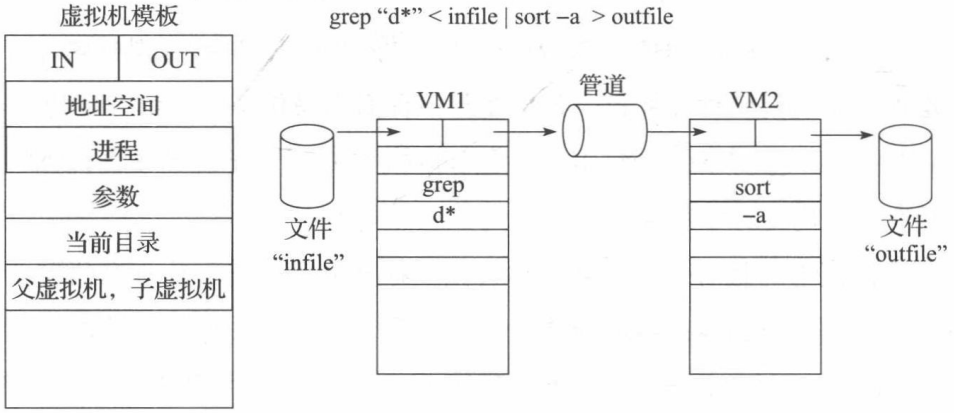


图10.7 Multics和UNIX两种操作系统引入了一种虚拟机模型，用于执行用户的命令。一个虚拟机模版（右图）提供了一个IN端口、OUT端口、指向地址空间的指针、指向地址空间中进程的指针、一组传递给进程的参数、虚拟机的当前目录以及一组指向父虚拟机或子虚拟机的指针。当用户在shell程序的界面上（右上角）输入了一条命令后，shell程序将这条命令解析为一组构成成分，并且创建一个虚拟机流水线去执行这条命令（右下角）。在这条命令中，搜索程序“grep”从文件“infile”中获得输入数据，并且仅输出那些以字符“d”开头的数据行。这些输出数据然后又被输入到排序程序“sort”中，该程序对所有的输入数据行按照字母顺序进行排序，并将排序结果输出到文件“outfile”中。通过对虚拟机的形式进行标准化、让任何文件都可以作为输入或输出以及使用管道对不同的虚拟机进行连接，这些系统仅使用一种简单的命令语言就能实现丰富多样的功能

对象

对象是一种虚拟机结构，来源于20世纪60年代称为“数据抽象”的编程实践，并逐渐演化出目前存在的120余种复杂的“面向对象语言”。对象是一种抽象实体，可以通过一组预先定义的操作对其内容进行查询或修改。对象的内部结构是对外隐藏的。如上文所述，文件是一个比特数据序列的容器，仅能通过打开、关闭、读、写等操作对其进行查询或修改；文件的内部结构是一组散落在外部存储器中的记录。

一些对象语言将对象视为数据结构，且仅能通过与其相关的一组过程对其进行操纵；这些语言的典型实例包括Python、Java、C++等。其他对象语言则将对象视为自主实体：这些自主实体可以接受外部的请求消息，并且通过响应消息将结果返回给请求方；这些语

言的典型实例包括 SmallTalk 和 Squeak 等。在分布式计算系统中，也通常将对象视为可以接受和发出消息的自主实体。 213 ~ 215

具有相似属性和相同操作集合的一组对象被称为一个类（也被称为类型）。一个文件就是由所有文件形成的类的一个成员。一个被称为类管理器的抽象机器可以被用作实现类操作的操作环境。例如，文件管理器可以打开、关闭、读或写任何一个文件，并且可以定位到一个文件在存储器中的所有成分（见图 10.5）。

对象类通常被组织为一个继承关系层级结构，使得一个对象具有的属性被传播到其子类对象中。例如：字节文件可能是比特文件的子类；目录可能是字节文件的子类，其文件内容是一组字符串和一组文件句柄之间的关联关系。

一个对象通常关联一个互斥锁，从而保证在一个时间点上只有一个进程能够使用该对象。这样做的目的是防止在并发环境中可能出现的竞争条件（即，多个进程同时读写一个对象中的数据，从而使得最终的数据状态存在不确定性）。 216

对象在很多时候被认为是一种首要设计原理，因为其很好地体现了“抽象”这种基础性原理。然而，对象实际上也是一种非常先进的概念，因为其提供了一种统一的方式去处理结构和同步这两个方面的问题。新手程序员会觉得对象这个概念不好理解，这主要是因为他们还没有理解虚拟机、信息隐藏、继承、同步等基本概念。

客户端与服务器

客户端 – 服务器模型提供了在分布式网络计算系统中对不同进程之间的交互进行组织的简洁方式[8]。服务器是一个能够响应特定服务请求的进程。客户端则是一个可以发出服务请求的进程。在一个网络中，客户端和服务器通常（但不是绝对）被不同的硬件机器所扮演。客户端和服务器之间的请求和响应数据以消息的形式在网络中传播。例如，一个网络文件服务器存放了该网络所有用户的文件，在用户工作站上的客户端进程发出请求去读写文件。一个认证服务器与用户工作站上的登录客户进行交互，在用户登录过程中对其身份进行认证。一个 web 服务器与客户端浏览器进行交互，将所请求的 web 页面传递给客户端。

客户端 – 服务器的思想很简单，但其实现则往往非常复杂。设计者需要应对关于通信、错误控制、同步等很多非常细节的技术问题（Birrell 和 Nelson 1984）。

在一些系统中，相互交互的两个进程之间的客户端或服务器角色可以灵活变化。在这些情况下，这两个进程被称为对等（P2P）进程。很多网络服务都被组织为这种方式。例如，互联网 TCP 协议会运行在网络中每一台计算机的一个本地进程中，本地的 TCP 进程可以向远程的另一个 TCP 进程发出连接请求，也可以接受来自远程 TCP 进程的连接请求。

总结

设计在计算领域的发展历程中始终是一个核心议题。计算领域第一代的设计者花费了很大的精力去实现高效、可靠的自动化计算过程。他们为我们留下了一个非常好的设计方案，即存储程序计算机（又被称为冯·诺伊曼体系结构）。这种设计方案至今仍在使用。

机器的指令集是存储程序计算机的用户接口。程序员设计出算法，并将算法用指令集
217 进行编码。从一开始，程序员就发现他们使用了大量的时间去寻找程序中的错误；为此，他们发明了调试技术。他们发现，程序设计是一个非常复杂且易于出错的活动。于是开始探索如何才能设计出可靠、可用、安全（DRUSS）的程序，这种探索至今仍在持续。经过多年的努力，他们开创了程序设计行业和软件工业，其中涉及了关于错误预防或确认的大量技术手段。即使已经存在了丰富的技术积累，程序中的错误目前仍然是一个严重的问题。

设计不仅仅是通过对硬件成分和指令的组织来解决问题，更是关注如何为计算机的用户产生价值。设计者应该去理解用户会如何使用程序，去理解什么会使用户感到高兴或愤怒。设计者应该掌握足够的设计技巧[9]。

在寻找设计优秀软件的系统性方法的过程中，设计者关注于5个成功的设计准则：需求、正确性、容错性、时效性、适用性。需求关注于系统行为的准确描述；正确性关注于如何在构造过程中防止错误的发生；容错性关注于在错误被修正/去除之前最小化错误的后果；实效性关注于如何对系统进行配置从而保证计算结果的准时性；适用性关注于用户满意度。

在设计计算系统和软件的实践中，人们探索出5种有效的设计模式：层级式聚合（体现在抽象、分解、模块化、信息隐藏中）、封装、级别、虚拟机和对象。这些模式被实现在语言、应用程序、操作系统的结构中。虽然不是灵丹妙药，但人们普遍认为这些结构有助于实现上述5种设计准则和更一般性的DRUSS目标。

由于篇幅限制，我们不能对其他更多的软件设计原理进行介绍。即使对于项目管理、错误限制、容错、网络结构、操作系统结构、正确性等问题，我们所讨论到的设计原理也仅仅是非常小的一部分。设计是一个具有丰富内涵的领域，对于决定一个计算系统的成败
218 具有非常关键的作用。如何进行优秀的设计可能是计算领域中最大的一个挑战。

第 11 章
Great Principles of Computing

网　络

人脑与计算机将会非常紧密地结合在一起，由此产生的合作关系将以大脑未曾有过的方式思考，并以一种当前信息处理设备未曾达到的方式进行信息处理。

——J. C. R. Licklider

基于冗余设计的数字计算机新技术使得廉价不可靠的链接变得潜在可用，网络从一开始就是专门为数据传输和生存力而设计的。

——Paul Baran

随着网络的增长，我们可能会看到“计算机设施”的广泛出现，就像如今的电力和电话设施一样，在全国各地为家庭和办公室服务。

——Leonard Kleinrock

很难想象，目前一个已连接世界上所有计算机的网络在 1960 年时还仅仅是一个梦想。在 20 世纪 60 年代中期，美国国防部朝着这个梦想迈出了第一步。它开始筹划名为 ARPANET 的资源共享试验项目，并于 1969 年开始在两台主机（host，网络术语，指连在一起的计算机系统）上运行 ARPANET [1]。

ARPANET 是演化为如今互联网的技术链中的第一个项目，因为初始的设计不能扩展为大型网络，可扩展“互联网”（或者说网络的网络）的想法在 1973 年才被引入。经过 10 年的测试和提升，互联网的设计成为官方标准，而几乎没人注意到 ARPANET 在 1989 年被废止了。万维网在 20 世纪 90 年代早期就覆盖了互联网，网络规模的增长非常显著，1981 年接入的主机数大约为 200 台，1990 年增长到了 20 万，2004 年为 2 亿，而 2014 年为 10 亿。如今的互联网包含数以百万计的分支网络，这也是网络可以为它的成员带来巨大价值的一个有力证明 [2]。

这一卓越的技术成就是设计者巧妙结合了第 1 章中所有 6 类计算基本原理的结果。互联网的设计使得其规模在 30 年间增长了 7 个数量级，且仍保持可靠的运行。作为一个案例研究，我们将回顾互联网的每个主要部分，并说明每一部分遵循的是哪一条计算原理。

网络的概念比互联网的历史要悠久得多。在 1750 年左右数学家欧拉（Leonhard

Euler）第一次通过画网络图证明了在每座桥都只能走一遍的前提下，无法把柯尼斯堡（Königsberg）的七座桥都走遍。从那时开始，图论成为数学的一个主要分支，用以研究物体之间的关系。网络包括节点（也称为顶点）集合和连接其中一些节点的链接（也称为边）集合。链接通常代表实体（例如信号、消息、商品、请求和承诺）从一个节点移动到另一个节点的路径。电气工程师发现了流体、能量、电流和电压守恒定律，并将这些定律应用于设计具有预期信号转换功能的电气网络。工业工程师和运筹研究人员开发了多种类型的网络模型系统，包括制造、运输、库存、通信和队列；网络模型使得能够对这些网络系统进行准确的性能预测。如今社会学家也使用网络来映射一个群体内成员之间的通信关系，并对权力和影响力进行推理。互联网展现了这些网络的不同侧面：它是一个通信系统、消息的传输系统、数据的信息检索系统、服务的队列系统、存储子系统的库存系统和社交网络的支撑系统。

弹性网络

在 20 世纪 60 年代早期，国防通信工程师开始寻找电话网络的替代方案。国防官员对于使用电话网络进行通信非常担忧，因为电话网络是基于电路交换原则的，这意味着在拨打一个电话号码时会使得一系列的机械开关打开，从而提供从呼叫基站到接收基站的直
220 接电路连接。电话网络已经演化为一个由“干线”连接的区域交换中心的集合，区域交换中心与本地交换中心相连，从而进一步与家庭或者办公室相连。这一网络在遭遇自然灾害（例如火灾或者洪水）或者人为破坏（例如恐怖袭击）时可能会被严重损毁，因为破坏网络中的一个或多个节点就可能完全损毁网络。这一网络不仅不安全——一条电话线可以很容易地被窃听——而且在面对敌对行动时也是不可靠的，因此电话网络被视为国防中的一个严重弱点。

Leonard Kleinrock（1961，1964）和 Paul Baran（1964a, b）首先发现了上述电话网络的替代方案。Kleinrock 阐述并分析了一个消息交换系统的随机模型。Baran 提出了电话通信的一个新设计：模拟语音可以被数字化，产生的比特流被分解为数据包，经过存储转发系统到达目的地，然后被重新组装为原始比特流[3]。一个数字文件也可以类似地将比特聚集为数据包。网络本身包含很多冗余路径，使得数据包在网络某些部分损坏的情况下可通过另外的路由到达目的地。可配置的路由器负责处理路由，它们接收消息，然后将消息转发到离目的地更近一跳的路由器。现代网络是两种结构的组合，包括面向长距离主干网的网状结构和面向本地连接的中心辐射型结构（见图 11.1）。局域网可以有多种形式（见图 11.2）。

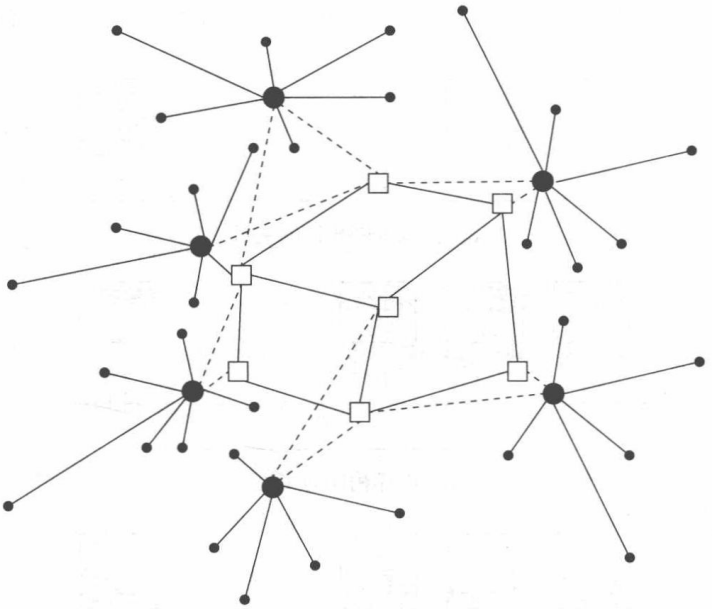


图 11.1　工程网络（例如电话网络或者互联网）包括很多类似这幅高度简化的图中那样的链接。一个主干网络（白色方框）为局域网（黑色大圆）服务，局域网与桌面电脑、服务器或移动设备（黑色小圆）等计算机相连。局域网通过多个链接（点线）与网络相连。主干网络被设计为网状结构，来保证单一主干节点的丢失不会将网络分割成孤立的、未连接的碎片。主干网络和局域网节点用数据包路由器实现。在互联网中，局域网被称为 LAN（本地区域网络），主干网络被称为 WAN（广域网）。LAN 和 WAN（黑色大圆）的连接点用网关路由器实现。网关路由器将数据包由 WAN 信号格式转换为 LAN 信号格式

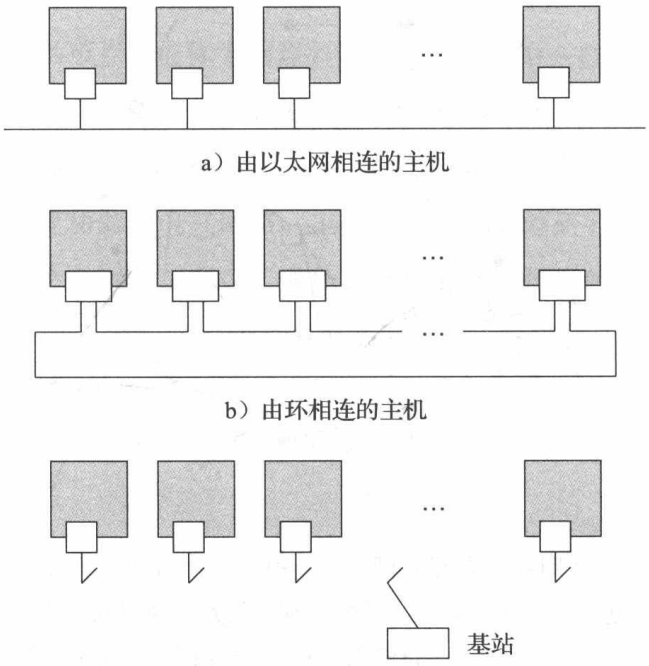
数据包交换

一个*数据包*是一个比特序列，其中包含了一些信头和一个数据块（见图 11.3）[4]。*数据包交换*是一种网络运行模式，它使得如图 11.1 中的主干网中称为路由器的子网设备将数据包运输到它们的目的主机。接下来的讨论将分析数据包网络的一些原理。

数据包交换是之前章节提到的多路复用原理的一个实例。多路复用意味着将资源分为块，然后将块分配给不同的个体。例如，主存是通过将其分为页来实现多路复用的；页可以被快速放置在方便使用的地方，当不被用到的时候可以被移除。磁盘是通过将许多文件分为记录并允许记录分散在磁盘上实现多路复用的。信道是通过将每一个信号流分解为数据包并在信道上传输数据包来实现多路复用的；同时多个信号流可以共享同一信道。

数据包多路复用是 20 世纪 50 年代电话系统中使用的时分多路复用中的一种。例如，语音通常是低带宽的，被数字化并压缩为数据包的模拟语音信号可以在所需时间的一小部分时间内就被发送，因此电话工程师得以通过单根线缆就可以无损或保真地获得多个数字化

通话。



a）由以太网相连的主机

b）由环相连的主机

c）由无线相连的主机

图 11.2 一个局域网（LAN）通过短距离链接与一组计算机相连。局域网在家庭、办公室和建筑中用到，其中一台计算机充当互联网的网关。以太网配置（图 a）将所有的计算机连接到一个将数据包发送到目标计算机的共享交换机上。环配置（图 b）将数据包从一个计算机环形传递到下一个计算机。无线配置（图 c）使用无线电信号以及 802.11 协议通过基站来交换数据包。在这些配置中都不需要路由器，因为每一台计算机都能看到所有的数据包但只选择发送给自己的。然而，能够看到所有数据包是一个安全弱点，因为一台不怀好意的计算机可能监听其他的数据包或者发送欺骗数据包

来源域	到达域	长度域	数据域

图 11.3 在数据包交换中，源站的数据流被分解为块然后插入数据包中。如图所示，在这个互联网数据包的简化视图中，来源域包括发送站的地址，到达域包括接收站的地址，长度域表明了数据包的总大小。数据域包括来自原始数据流的一个数据块，这个数据块可能包括一个序列号，使得接收协议能够将数据块按照它们原来的顺序重新组装。数据包从一个路由器被运到另一个路由器，在每一跳中逐步接近接收方。如果一个内部网络节点或者链接发生故障，路由器会进行重新配置，将数据包通过其他路由发送

数据包网络吸引人的原因主要有三点：第一，它同时迎合了不同流量类型的需要。例如，语音流量是固定速率（通常采样率为每秒 44 000）的持续很长时间的比特流。但是在网络服务器向发出请求的客户端发送页面时，网络流量通常是短暂而高强度的脉冲。数据包网络在传输数据包时，并不考虑数据包中是持续的语音还是脉冲式数据流。

221 ~ 223

第二，数据包网络天然就比基于电话的数据网络高效。电话网络通过两个步骤处理呼叫：

1）拨号以建立回路。

2）在回路上传输。

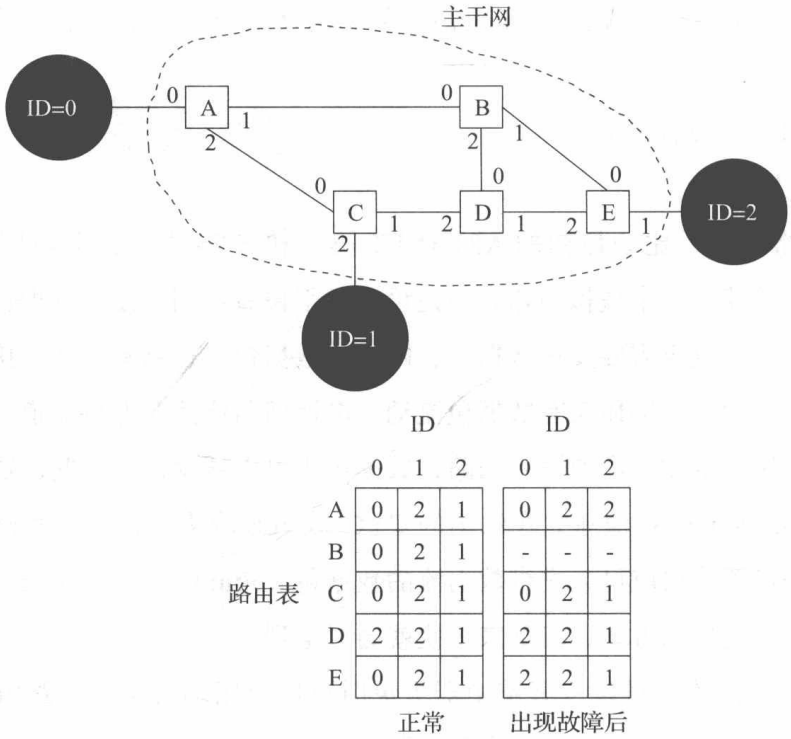
早期的网络机制，比如最初的 ARPANET 协议和 X.25 协议，会仿照电话网络的做法，在传输数据前建立一个虚拟回路。但是设计师很快意识到，使用数据包就不需要建立保持的回路。发送方仅仅需要发送数据包，网络为数据包进行路由，然后接收方在数据包到达时做出响应。此外，即使没有数据包流动，电话回路依然会保持通道打开。但是数据包网络只要有可用的带宽，即使没有回路，数据包也可以被发送。因此，数据包更好地利用了带宽。Louis Pouzin 在他建立在法国的 CYCLADES 网络中使用了术语*数据报*来描述这种不需要预先设置协议就可以携带数据流的数据包。Vinton Cerf 和 Robert Kahn 在 TCP 协议中设计了一个类似的方案，接下来很快就会讨论到。

第三，数据包网络在链接或者路由器失灵时可以启用动态重构。路由器会接收数据包并转发到离目的地更近一跳的下一个路由器。如果一个路由器检测到转发路径或者下一跳的路由器损坏了，它可以从另一条路径发送数据包。偶尔会遇到数据包在传输时链接或者路由器损坏从而丢失的情况。但这并不是一个问题，因为传输协议会检测到数据包丢失并重新发送它们。

224

路由器在一个队列里存储接收到的数据包，然后使用路由表将它们转发到离目的地更近的下一个路由器。产生路由表的路由算法负责寻找到达目标主机的最短路径（由路由器之间的跳数衡量）。路由算法通常在后台运行，因此路由表会被持续更新以表示当前网络的连通性（见图 11.4）。Edsger Dijikstra（1959）被认为是如今用于寻找节点间最短路径的高效算法的发明者。当路径长度使用延迟或者开销而不是跳数来度量时，也会使用其他算法来计算最短路径。

因为网络采用存储转发的设计，路由器成为潜在的交通拥塞点。如果涌入一个路由器的流量使它的队列溢出会发生什么呢？一种可能是丢掉数据包，并让传输协议检测到这一点然后重新发送它们，如果丢失率过高，则进行传输回退。一种不太常见的可能是使用链接协议阻止一个路由器发送数据包，除非接收方有空间能够接收。这种策略防止了数据包丢失的情况，但是增加了主机与网络的接入点的拥塞，降低了它们的传输速率。后一种策略在链接管理协议中被称为“流量控制”。因此毫不奇怪，队列网络模型（第 9 章）对于预测存储转发网络的平均产出率和传输时间非常有效。



路由表

	ID 0	ID 1	ID 2
A	0	2	1
B	0	2	1
C	0	2	1
D	2	2	1
E	0	2	1

正常

	ID 0	ID 1	ID 2
A	0	2	2
B	-	-	-
C	0	2	1
D	2	2	1
E	2	2	1

出现故障后

图 11.4 这个例子展示了一个主干网与编号为 0，1，2 的三个局域网主机相连的情况。每一个主干路由器（正方形框）有三个可以发送或者接收数据包的端口（编号为 0，1，2）。正常的路由表（下方左边的一栏）会告诉路由器对于一个有着特定目的地的数据包应该选择哪条链接。例如，从主机 0 到主机 1 的数据包会走路由 A,C；从主机 0 到主机 2 的数据包会走 A，B，E。如果路由器 B 出现故障，重构算法会修改路由表使所有流量绕开 B（右下方），现在从主机 0 到主机 2 的数据包会走 A，C，D，E。注意到如果 A，C 或 E 出现故障，会有一个主机与主干网断开。如果 D 出现故障，从主机 2 到主机 1 的数据包会走 E，B，A，C

互联网络协议

数据包交换和可重构性原理不足以使一个网络可用。互联网包含数亿主机和数以百万计的局域网。用户对于这些主机提供的服务感兴趣，那么一个用户的进程是如何寻址到目标服务器，从而可以发送请求服务的数据包并接收响应的数据包呢？寻址方法不应该依赖于局域网的变幻无常或者网络上可用的路由器。如果没有统一的寻址服务的方法，局域网之间的差异以及可用服务的变动会使得网络变得混乱从而不可用。

Vint Cerf 和 Bob Kahn 在 1974 年针对这一问题提出了一个解决方案。他们提出了一个非常大的地址空间，其中的地址有着足够的位数使得互联网中的每一台计算机都有自己唯一的地址。他们设计了一个叫做 Internet Protocol（IP）的协议，给定某服务器的唯一地址就能定位到它，这些唯一的地址称为 IP 地址。

225 ~ 226

一个 IP 地址有 32 位，被表示成 4 个 8 比特的字节。一个字节可以编码一个 0 到 255（2^8-1）的整数。IP 地址的标准记法包含四个由点号分隔的整数（从 0 到 255），例如 192.168.3.55。IP 地址的总数是 2^{32}，或者说大约 40 亿。随着移动设备、家电以及任何包含嵌入式计算机的设备的普及，互联网快速增长，如今出现了 128 位的 IP 地址。

许多局域网包含了一项叫做*动态主机配置协议*（DHCP）的服务来减少网络中对固定 IP 地址的需求。当一个计算机启动后，它从一个叫做 DHCP 服务器的本地计算机那里请求 IP 地址，DHCP 服务器会从本地子网地址池中为其分配一个临时 IP。因为通常实际登录的主机数量比所有连接到局域网的主机数量少得多，这一策略使得一个给定的 IP 地址集合能够被一个大得多的主机集合所共享。

IP 协议不保证一个数据包会到达它的目的地。在路由过程中很多事情都可能出错，例如，一个路由器可能不知道一条被选中用于转发数据包的链接并不工作；涌入一个路由器的数据包可能因为路由器的缓冲区已经被其他数据包填满而被丢弃；链接上的噪声可能会篡改数据包中的某一位，使得接收路由器无法对信头字段进行解码。这些或者其他错误可能导致数据包在到达目的地之前丢失或者损坏。IP 协议之所以被称为一个“尽最大努力的数据报协议”是因为它在数据包没有遇到错误时将数据包当做数据报发送，并且不会尝试重新发送丢失或者损坏的数据包。

Vint Cerf 将 IP 协议与邮政服务中只处理明信片的服务版本做了类比。一张明信片就像一个数据包，包含到达域、发送域以及一个存放消息的有限区域。当你邮寄一张明信片时，你知道它可能不会被送达，或者可能在运输中被损坏；你不知道投递会花多长时间。被邮寄的一系列明信片也不保证它们会按照发送的顺序被接收：每一张明信片可能会被不同的邮政车或者飞机按照不同的路线运输。

尽最大努力的数据报投递的原则是简单有效的。它将数据块封入一个 IP 数据包中从而数据包可以起到数据报的作用（Cerf and Kahn 1974，Cerf 等 1974），这被运用在 CYCLADES 网络（Pouzin 1973，1974）以及 TCP 协议的最初设计中。1977 年数据报功 227
能从 TCP 中被剥离成了单独的 UDP 协议。TCP 和 UCP 是运行于 IP 协议之上的并行服务。IP、TCP 和 UDP 在 1981 年的规范形成了如今标准的基础。

传输控制协议

对于很多应用我们需要一个比 IP 更复杂的协议，从而可以在一个可能丢失数据包的网络中以很高的投递准确率来运输数据。由 Vinton Cerf 和 Robert Kahn（1974）设计的传输控制协议（TCP）完成了这一目标，它使得文件能够被可靠投递，即使携带文件的数据

报是不可靠的。

TCP使用端到端的纠错原则在丢失网络上获得可靠性（Saltzer等1984）。这一想法是让接收端发送“确认数据包”给发送端，从而可以判断出有多少数据流已经被成功接收。由于确认数据包可能丢失，发送端如果在一个设定的期限内没有收到确认数据包，它会自动重新发送数据包，同时接收方会忽略由于重发造成的重复数据包。这一设计在连接结束时检测和替换丢失的数据包，并不需要假定网络上正在发生任何查错或纠错的操作。当然，网络中的纠错机制（例如链接中的汉明码）降低了丢包率，提高了TCP的传输速率。

TCP的操作准则可以继续借助Vint Cerf关于邮政服务的类比来进一步理解。发货人在只使用明信片的情况下怎样发送一本书呢？如果托运人和接收人已经对于在发送前如何拆解一本书以及发送后如何重新组装一本书达成了一致，那么就可以做到发送一本书。发送方将书页拆分成小的碎片并将碎片贴在明信片上，然后用序列号标记明信片并保存所有明信片的备份。接收方将收到的明信片按合适的顺序（由序列号决定）放置，然后取下明信片上的碎片将它们恢复成一本书。

为了告知发送方知道哪些明信片已经被接收，接收方每一次返回一个确认（ACK）明信片概述目前有哪些明信片已经被接收。发送方收到ACK后，会丢掉所有已被确认接收的明信片的备份。然后是重要的一步：如果发送方在一个超时限制内没有收到ACK，它
228 会自动重发明信片，重复这样的操作直到收到这一明信片的ACK，或者直到发送超时并报告发送失败，这一方法克服了网络中ACK可能丢失的问题。接收方会忽略来自发送方的重复明信片。

这一设计在面对一本真实的纸质书时可能会显得很笨拙和混乱，但是面对数字文档，这一设计是简单、干净且快速的。TCP在一个有噪声的网络上会花费更长的时间，因为重发和确认降低了它的速度，但是只要一个数据包到达的概率不是0，TCP最终会传送整个文件。

客户端与服务器

TCP使得大量网络服务成为客户端与服务器的形式。客户端与服务器是它们各自主机操作系统上运行的自主且持续的进程。系统的一个客户端进程通过TCP与另一个系统上的服务器进程相连接。通过这个连接，客户端产生请求，服务器端作出响应。

TCP也可以用于对等网络的交互，即两端的进程都可以发送请求和做出响应。

与网络接口相连的进程依赖于本地操作系统的多路复用来处理进入的数据包，并将

这些数据包传递到它们相对应的服务进程。这些接口也可以处理来自外部同一主机的多个连接，例如一个用户同时访问两个不同的网页。

TCP 通过端口来支持到特定进程的连接。*端口*是一个客户端或服务器进程的本地名称（Pullen 2000）。例如，一个主机的网络服务器被分配到 80 端口。一个寻找主机 H 上网页的客户端将请求打包为一个数据包，在数据包中它请求 TCP 将其投递到“H：80 端口”，主机 H 上的接收器则将数据包传递给与 80 端口连接的本地进程。主机地址与端口的结合（“H：80 端口”）称为*套接字*，TCP 被设计为将数据包发送到套接字而不仅仅是主机。随着时间的推移，成千上万的服务已经被定义并分配了端口号。

文件传输协议（FTP）在一个更加复杂的场景里阐述了客户端 - 服务器的关系。FTP 被分配到 21 端口，它是为在两台主机上都有账户的用户设计的。主机 A 上的用户调用本地 FTP 程序并请求 FTP 打开到套接字“B：21 端口”的 TCP 连接。主机 B 上的 FTP 服务器向主机 A 上的 FTP 请求用户的登录凭证。一旦登录，用户切换到期望的目录并发出包含本地和远程目录名称的“put”命令。一旦文件被完全接收到远程目录，主机 A 上的用户会将主机 B 的确认视为一个完成消息，这时用户就可以退出本地主机 A 的 FTP 程序，关闭与远程主机 B 的 FTP 会话进程。 229

TCP 和 UDP（User Datagram Protocol）不是客户端 - 服务器系统。它们只是连接两台主机上的诸如客户端与服务器或节点的进程通道。

域名系统

上文描述的 TCP 和 IP 协议使用 IP 地址来识别数据包的发送方和接收方。32 位 IP 地址（IPv4）以一种人类更可读的方式描述：四个字节中的每一个都以一个 0 到 255 之间的整数表达。例如，IP 地址 10000001101011100000000100011100 被表示为 129.174.1.28，这些数值表示比 10 位电话号码更加难以记忆（对于十六进制表示的 128 位 IPv6 地址，情况会更糟）。因此互联网采用了更加容易记忆的符号形式的主机名称系统。例如，“gmu.edu”的 IP 地址是 129.174.1.28，同时“gmu.edu”比它的 IP 地址要容易记忆得多。

我们可以在一个域名服务器的帮助下将符号形式的主机名翻译为 IP 地址。域名服务器像电话簿一样工作。在互联网上，符号形式的主机名被称为域名，域名服务器是域名服务（DNS）的一部分，向 DNS 请求查找一个域名，然后它会以对应的 IP 地址来响应。

TCP 软件被设置为自动请求将域名翻译为 IP 地址。例如，用户请求“发送文件 F 到 gmu.edu”会通过与发送主机网络相连的域名解析器翻译为“发送文件 F 到 129.174.1.28”。DNS 并不是以单个服务器实现的，它是一个由服务器和缓存组成的网络，

缓存可减少在查找域名时产生瓶颈和拥塞的几率。

互联网的设计者仔细考虑过如何产生独一无二的域名。最初的 ARPANET 在 SRI 国际有一个主文件，其中保存了所有的域名和它们对应的 ARPANET 地址。对于不断
230 增长的互联网来说，给所有主机分发这样一个文件会成为一个瓶颈。在 1983 年，Paul Mockapetris 设计了 DNS（用于域名注册的分布式服务），DNS 使域名到 IP 地址的翻译过程自动化，并允许对域名空间的任意扩展。

在 DNS 中，符号名被点号分隔为字段，就像“www.gmu.edu”中的那样，最后一个字段是顶级域，代表了工业、政府、组织属群或者国家代码。一个独立的机构——互联网名称与数字地址分配机构（ICANN）负责管理顶级域名[8]。在 1985 年，其仅仅包含一些顶级域名例如 .com, .mil, .gov, .edu, .org 和 .net。在 2014 年，这一数字已经增长到大约 350 个（包括国家代码），另外还有 2000 个域名在考虑之中。

每一个顶级域名有一个登记处负责将域名分配给域内的组织；每一个组织也有一个登记处负责将域名在组织内分配；以这种登记处层级结构分配的域名由从低到高的注册项联系在一起。以域名 www.cs.gmu.edu 为例：

- “www”由乔治梅森大学计算机系的登记处分配。
- “cs”由乔治梅森大学的登记处分配。
- “gmu”由 .edu 域的登记处分配。
- “edu”由 ICANN 分配。

这一方法将分配域名的权利下放到本地组织，从而可以选择意义更加丰富的域名并且降低了瓶颈和拥塞。注意，域名是不能独立于地理位置的；例如 cnri.reston.va.us 属于位于美国弗吉尼亚州雷斯顿的国家基础设施公司，而 inria.fr 属于法国的国家信息与自动化研究所。

域名映射是动态名称翻译原则（第 7 章）的一个实例，这一原则中系统将处于较高层级的名字自动翻译为处于较低层级的名字。在域名映射中，这一原则具有两个主要好处：第一，具有将任意 IP 地址分配到任意域名的能力；第二，在不改变服务器域名的情况下可以改变服务器 IP 地址。在一些地址翻译系统中，低层级的名字在较高层级中是被隐藏的——但在互联网中并不是这样，IP 地址并没有对那些想要使用它们的人隐藏。域名解析系统通过缓存本地域名服务器上 DNS 数据库的一部分，避免由于将消息发送到更远、
231 更容易发生拥塞、更集中的域名服务器上而造成的消息延迟，来提高它们的性能。相同的方法对于互联网上许多可访问的服务都是有效的。例如，如果不是因为叫做内容分发网络（CDN）的缓存网络，许多热门服务会成为瓶颈。

网络软件的组织结构

上文的描述表明互联网是一个十分复杂的大型系统，它的真实大小只能被估计，它拥有 10 亿主机和上百万局域网。它的路由器、链接、服务器、应用、数据中心、服务提供商等等的数量是如此的巨大以至于《经济学家》杂志在 2012 年声称整个互联网消耗了全世界约 3% 的电力；国际能源署（iea.org）推测 2014 年这一数字大约是 6%。国际经济的很大一部分也开始依赖于互联网，将其作为交流和做生意的一种途径。

互联网能够取得如此多的成就，无处不在并且工作可靠，该能力是对其设计师远见卓识的证明。他们利用了“分层原则”的设计，以一种能够扩展到大型网络并且被大多数人理解的方式使组件结构化。下面让我们简单回顾一下互联网软件的组织结构。

在讨论 TCP/IP 时，我们使用了将书通过邮政服务进行投递的一个类比例子。现在回顾整个过程，看看所有的组件如何组装为一个有效运转的系统。

- 希望给朋友邮寄书的顾客，将书连同朋友的地址交给托运人。
- 托运人将书页切成碎片，把碎片贴在已经编号的明信片上，做好备份，并将原件交给邮局。托运人等待确认性的明信片以了解何时可以丢弃备份件。托运人经过一个超时范围如果还没有收到确认，会重新发送明信片。
- 邮局将明信片打包在邮袋中并在邮袋上标记路线和邮递员。
- 邮递员将邮袋放在卡车或者飞机上，将邮袋送到目的地。
- 在目的地，邮袋被取下卡车并送到邮局。
- 邮局取出明信片，将它们按序号排序后发给托运人。 232
- 托运人从明信片上取下书页碎片并将它们重新拼好，发送确认明信片，偶尔可能会等待丢失的明信片。
- 托运人将完整的重新组装的书交给朋友。

图 11.5 展示了邮局中的层级与互联网软件中层级的对应，在互联网中发生的动作序列也是如此，文件会在层级中逐级下降直到它在物理介质中被编码为信号，然后在接收端逐级上升直到它被解码并重新组装为文件。

在这种结构中，发送端的每一层级可以视作自己在与接收端的同一级进行通信。例如，托运人将自己的工作视作把书送到接收端的托运代理。邮局将自己的工作视作把明信片送到接收端的邮局。邮递员将自己的工作视作把邮袋送到邮寄地点。发送端的 TCP 以相同的方式将它自己视作与接收端的 TCP 直接通信。尽管实际的通信流是先下降到物理层然后上升到接收端的 TCP，但表面上看，通信流是直接发送到接收端 TCP 的（见

233 图 11.6）。这种表面上的一个层级与另一主机上对应层级的直接连接是互联网软件的特点，它使得一个层级的设计者可以忽略软件底层的细节。

图 11.5 互联网协议软件按照一系列的层级排列（左边部分），这些层级类似于托运人将书投递给客户过程中使用的层级结构（右边部分）。某一层级的服务由较低层级的更简单的服务组合而成，TCP 层管理可靠的数据流传输，IP 层管理数据包的“尽最大能力”投递，链路层在路由器之间选择路由并进行简单的纠错，信号层在选定的链路上将数据包编码为信号

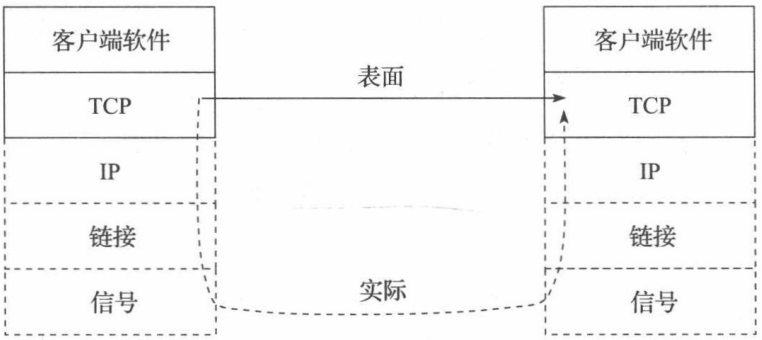


图 11.6 同样的互联网协议软件被放置在网络中的每一台主机上，软件的层级造成了一个现象：每一级都在和另一台主机上对等的层级直接通信，但表面上的连接是由较低层级实现的。例如，TCP 层将一个文件以一系列数据包的形式发送给 IP 层，IP 层使用动态重构路由器通过物理链接将数据包以信号的形式发送。在接收端物理信号被解码为数据包，数据包从本地 IP 层流入到 TCP 层，TCP 层将数据包重新组装为文件。网络工程师将软件层级的集合称为协议栈，因为它们被实现为堆叠的软件模块

万维网

1989 年，Tim Berners-Lee 在瑞士的欧洲核子研究实验室为了支持研究论文和其他数字文件的交换（Berners-Lee 2000），第一次提出了万维网（WWW），其在 1991 年底首次上线。在 1993 年，伊利诺伊大学香槟分校的国家超级计算应用中心发布了万维网的第一个图形化界面——Mosaic，突然间许多人能够在互联网上进行自动信息分享。万维网如同野火一样兴起，互联网“.com”公司的时代开始了。如今万维网被视作互联网的信息分

享层。任何形式的数字对象，包括文字、图片、声音、音乐和视频都可以在一个万维网系统中得到访问、传输、展示或者播放。 234

万维网技术来自于一些现有技术的聚合，用于解决信息共享问题：

- 互联网——尤其是支持客户端－服务器应用的 TCP/IP 协议。
- 域命名——互联网中主机唯一的、有层级的名字。
- 超文本——以非线性结构相连接的文本块构造的文档。
- 标记语言——使用标记对数字文档进行注解以告诉其他程序如何显示或者渲染文档。

这些技术之所以能结合起来是因为 Tim Berners-Lee 是所有这些技术的实践者，他的发明包括：

- URL——统一资源定位符，将主机域名连同文件在主机文件夹中的路径名连接在一起。由于路径名是分层的，因此 URL 也是分层的。对于互联网中数据对象来说，URL 是它们全局唯一的符号名。
- HTML——超文本标记语言，用标签对数字对象文件做注解以表明如何渲染数字对象的各个部件。例如，文件中的文本块可以被标记为标题、段落或者列表，然后显示程序就会以适当的方式显示它们。超链接将数字对象的 URL 进行编码，分布在其他被标签标记的元素之中。超链接可以是对其他文档的引用，同时也可以指向互联网上的任何对象，例如图像或者声音文件。
- HTTP——超文本传输协议，当光标选中一个超链接时会由浏览器自动触发。HTTP 会打开通向 URL 链接中主机 80 端口的 TCP 连接，然后将文件路径名传递到 HTTP 服务器，服务器会查找这个文件，将其副本传回发送端，然后关闭 TCP 连接。

在大众万维网出现不到两年时，出现了第一个网络搜索引擎，因为用户希望在互联网上找到所需信息。搜索引擎是一种输入关键字然后在互联网上搜索包含关键词的网页的服务，它会返回网页的 URL 以及匹配的部分文本。为了让搜索更快，搜索引擎会运行叫做“索引器”或者“网络爬虫”的子程序用于系统地遍历主机名，并打开与这些主机 80 端口的连接，找到所访问页面上的所有 URL，并继续访问这些 URL。它将每一个页面的 235 副本存到主数据库中，然后创建一个主索引，以发现给定关键词的 URL。从 1993 年起，已经出现了数十个网络搜索引擎。

在 2005 年前后，Google 成为了最大的搜索引擎服务。Google 使用网络爬虫维护一个持续更新的整个万维网的快照，并构建了一个能够在给定字符串下高速查找可能包含该字符串页面的索引。其他热门的搜索服务还有 Yahoo、Bing 和 Wolfram Alpha。

搜索引擎局限于它们能看到的东西，它们查询的是由爬虫填充的数据库。所有不是URL 目标的内容都不会被发现；需要用户登录的网页 URL 或者提供数据库检索界面的URL 都不会被收录；一些只能被诸如 FTP 或 TELNET 协议访问的服务器不是万维网的一部分。有时网络爬虫可能会陷在互联网的“岛”上——与网络中其他部分不相连的部分。网络专家相信被搜索引擎看到的网络内容最多占 10%。

即使搜索引擎几乎是盲目的，它们可以看到的网络内容数量仍然是惊人的。在 2013 年，万维网基金会估计网页的数量超过一万亿，即使搜索引擎只索引了其中的 10%，一个请求也需要查找一千亿条记录。这就是为什么对于一些关键词集非常容易得到一个压倒性数量的“命中”纪录。例如，在 9/11 恐怖袭击之后，关键词“ Osama bin Laden”产生了近 300 万次的命中纪录，没有人有能力查阅哪怕其中一部分的纪录。因此，人可以定位的网页数量其实是很小的。Hubert Dreyfus（2001）评论说，在网上找某样东西就像在一堆针里找一根针一样。John MacCormick（2012）将网络搜索比作在世界上最大的干草堆里搜索。

更加有挑战性的是几乎无法分辨获得的信息是否是查询的最佳答案，独立证实新的信息是非常困难的。然而，人们仍然认为网络搜索是有价值的，因为即使贫乏的信息也可以很有用。值得注意的是网络搜索可以持续发现有用的信息，更值得注意的是上百万的人花费数不清的时间建设网页供他人查找。

网络已经成为电子商务的媒介：在互联网上以电子方式进行商业交易。商业公司和其客户在发送数据前使用加密协议，从而提升安全等级和保护隐私。第一个加密协议——
236 1996 年的安全套接字层（SSL）协议在 1999 年被传输层安全（TLS）协议所取代。TLS 允许两个主机协商加密参数，然后对它们之间的流量进行加密；TLS 保护密码、支付信息、银行账户号、信用卡号等等在网络接口之间传递的部分商业交易数据。用户可以分辨出该协议何时正在被使用，因为一个锁的图标和字符串“https”会在他们浏览器的地址栏出现[10]。

除了新的协议，电子商务还给我们提供了新的商业模式，包括购物（在线商店和购物车）、企业间的协调（工作流）以及定价（eBay 拍卖）。伴随着这些新概念的发展，点燃了关于“新经济”的热情，导致了世纪之交的互联网泡沫破裂。

除了这些商业实践中的创新，互联网也激发了阴暗面的创新——垃圾邮件、病毒、蠕虫、劫持、拒绝服务攻击、入侵和其他肮脏的东西使得互联网的使用越来越难并且越来越危险。很难找到应对这些问题的技术方案，少数已经被提出的方案（例如互联网用户 ID）也遇到了激烈抵制。

互联网确实给了我们一个新的世界以及对于现实的新概念。有一天它有可能同其他的伟大发明一道被列入到 James Burke 的《宇宙改变的那一天》中。

网络科学

随着互联网络的增长，科学家开始对互联网中记录的人和对象之间的连接关系感兴趣。例如，研究者研究了有关文献引用的统计；组织分析师通过研究电子邮件关系的模式来发现组织里的上下级关系；执法官员将嫌犯的交谈与支付历史联系起来；反恐人员构建恐怖分子的网络以期摧毁他们。建模者来到互联网，开始挖掘关于关系的数据，当他们开始发现有趣的模式并做出令人震惊的预测时，他们开启了一个叫做网络科学的新领域。

Alberto Barabasi 将统计物理学的方法应用于从数据中得到的网络图上（Barabasi 2002）。例如在网络上，一个节点可能代表一个页面，而一个链接代表连接两个页面的一个 URL，节点的度是指接入链接的数目。Barabasi 发现这个数据满足“幂定律”，即度为 x 的节点数量 $P(x)$ 与 x^{-a} 成正比，其中 a 的范围为 1 到 3。他还注意到遵循幂定律的数据 237
是“尺度无关”的，因为对于任意 x，缩放 x 会使得 $P(x)$ 以相同的倍数缩放。例如，度为 $2x$ 的节点的数量为 $(2x)^{-a} = 2^{-a} \cdot P(x)$（见图 11.7）。另外，Barabasi 发现以偏好依附增长的网络是尺度无关的，偏好依附意味着一个新的链接与一个节点连接的概率等比于该节点的度，这在网络中通常是成立的：一个热门网站相比一个不出名的网站更可能吸引新的链接。

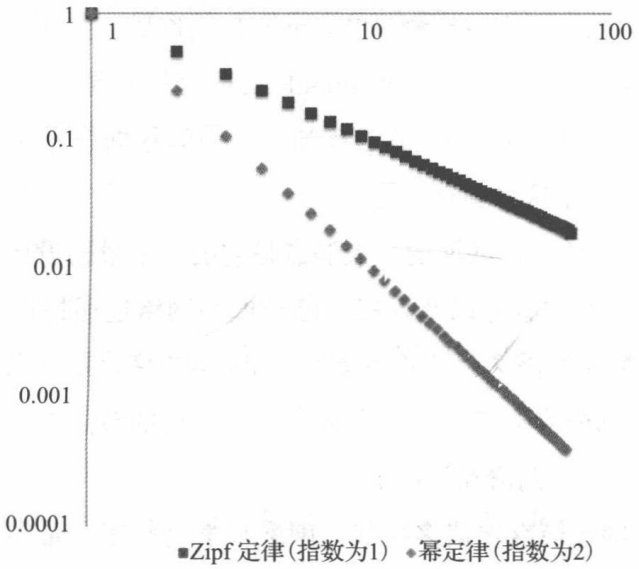


图 11.7　来自网络连接的节点度数据在对数 – 对数图上倾向于拟合为直线，这意味着度数为 x 的节点数量 $P(x)$ 正比于 x^{-a}，a 为常数。指数 $a = 1$ 时，这就是著名的 Zipf 定律，即 $P(x)$ 正比于 $1/x$，指数 $a = 2$ 在许多网络连接图中也很常见

尺度无关的说法应用到互联网的路由器连通性时，可以得到两个结论。一个是互联网对于随机节点故障具有高容忍度，因为大多数节点的度数很低，损坏其中一个，几乎不

能影响任何事情。另一个结论更加重要：互联网更易遭受严重破坏，因为网络模型表明高
238 度数节点的数量很少。这一弱点被称为互联网的“致命要害”，因为它暗示了一个攻击者可以通过找到并损坏一些枢纽来破坏大部分互联网。

致命要害的说法与互联网工程师自己的风险评估或与重大互联网事故的实际情况并不一致。2009 年一些研究者公开质疑了这一说法（Willinger 等，2009），他们注意到幂定律连通性的说法来自于 traceroute 的数据。traceroute 是一个互联网工具，用于返回探针数据包在移动到目的地过程中所遍历的 IP 地址列表。跟踪路由数据不代表任何实际的物理连接，原因在于协议栈：IP 层不能看到路由和链接层，而物理连通性在这些层中是可见的。

子网和主干链接的工程设计由两大因素驱动：为了避免连接丢失的节点和链接冗余性的需要与处理期待流量能力的需要。这些工程目标驱使工程师对偏好依附非常清楚，这就产生了像图 11.1 中那样的结构，局域网依附于主干网，度数越高的节点越倾向于在局域网而不是主干网中。Willinger 和他的同事（2009）批评了那些没有验证就接受“互联网路由器连通性是尺度无关的”说法的人们，互联网根本不存在尺度无关的致命要害。

换句话说，尺度无关的说法可能适用于网络图，但是对于按照网络图连接的物理工程网络来说并不适用。

那么对于社交网络、网页或者电子邮件网络的连通性数据呢？尺度无关的说法对它们适用吗？是的，因为这些数据直接测量节点和它们的度数。然而，在将这一结论转换到物理互联网时我们必须小心。例如，类似 google.com 这样非常热门的站点看起来是枢纽，因此非常容易被攻击。然而，Google 的工程师将它们的数据仓库和查询引擎实现为高度分布式的网络，恰恰避免了这一弱点。

Dark Clark（1988）相信互联网的最大弱点是它的一个最初设计：信任其他节点。恶意软件、僵尸网络、拒绝式服务以及入侵者已经成为网络连通性以及性能问题的最大来源。例如，互联网上最大的破坏性事件可能是 1988 年的莫里斯蠕虫，它使得当时相对较小的互联网中 10% 的部分瘫痪了数天；大部分停止工作的节点被它们的管理员自发断开网络来停止蠕虫的蔓延，直到清除掉蠕虫。

尽管对物理互联网建模存在诸多挑战，网络科学仍然很有希望构建更好的互联网模型以及设计不能被轻易损坏的核心互联网架构。

致谢

239 ~ 240

我们十分感谢互联网专家 Vint Cerf 和 Robert Beverly 对本章的细致评论。

第 12 章
Great Principles of Computing

后　记

本书的目的在于阐明计算机科学是建立在一系列基本原理之上的。这些原理告诉我们如何利用物质和能量来进行计算。这些原理帮助我们理解计算机可以做和不可以做什么以及计算机技术的限制在哪里。这些原理能够帮助我们找到技术和创新机遇的联系。这些原理能够帮助我们评估风险和避免计算机技术对人们的生活造成损害。这些原理是永恒的，即便在久远的未来，当现有的技术成为博物馆里的老古董时，这些原理依然是有效的。

我们希望以这个主题中一些主要经验的反思作为本书的结尾。

没有意识的机器

在撰写本书的时候，我们一直浸淫在计算机运作的原理之中。我们发现我们必须面对计算中无处不在的物质和能量。不管以何种方式，所有的原理都涉及如何控制物质和能量来进行计算。所有的事物背后都有物理原理的支持。每一个程序最终都是使用电路或者其他媒介来控制信号流。每一位设计人员都在探索能够把物质和能量转化为期望输出的程序。

我们相信，那句著名的标语“比特而非原子”使得人们低估了物质和能量在计算运行中的重要性。请注意，根据国际能源署估计，在 2014 年全世界将近 6% 的电力被用来支持因特网的数据中心和数据连接。即便如此，在过去的十年里，随着电子技术的飞速发展和几乎所有事物的全面电子化，本质化的改变已经发生。现在我们能够以极低的研发费用、免费的无错拷贝、世界范围的传播、非常便宜的销售价格等方式生产新的电子产品（例如“app”）。由此形成的电子经济是丰富的，而非稀缺的。这种转变并不是由于“比特”导致的，而是在于我们已经学会了如何廉价地控制大量的信号和原子。

尽管我们已经善于使用抽象手段来解释和设计计算，但计算本身无法抽象。计算是信号的运动和物质状态的转变。这些运动和转变的每一步都是纯粹的物理现象，遵守一定的物理定律。计算机的任何部分都不涉及智能。计算机的无思维性和机械性是非常令人惊讶的。

智能机器

那么计算机是如何做到看起来是智能的呢？对于我们而言，答案是非常简单的：计

算机是被设计成这样的。设计人员把软件和计算机本身塑造成能够产生他们所期望的反馈的样子。设计人员经常利用原型系统来学习用户如何对不同机器行为进行反应。如果他们不喜欢某一种反应，那么就更改相应的设计。从他们的角度来看，大部分用户不会把一个错误结果当作智能的疏忽，而是简单地认为机器坏了。当你正好碰上一个计算机的行为看上去是有智能的，其实是因为你正好按照设计人员所预期的进行反应，而机器本身并没有智能。

有一部分人认为由大量没有智能的机器所组成的大规模网络能够发展出新兴行为（emergent behavior），从而在事实上产生智能[1]。大量的“新兴行为”是有意设计的结果。例如，一个设计师可以在观察到先前用户对计算的反应后，循环地塑造计算的本地行为。在这些情况下，这些“新兴智能”其实是设计人员意图的反映。更普遍地，技术总是嵌入在社交系统中，它们遵循人类在系统中的行为。例如，看上去电话推销机器人正在“侵入”我们的家园，但事实上，公司选择这些机器的原因在于这些机器能够产生所期待的结果。这大概能够解释很多显然的新兴智能行为，而不需要假设这些机器或者网络具有智能。

计算机也可能由于其运行速度而被认为具有智能。尽管你可能也意识到需要通过数以十亿计的计算步骤来产生针对问题的一个答案，但计算机能够在眨眼之间给出反馈是令
242 人震惊的。今天的计算机比20世纪40年代的计算机快了将近10^{14}倍，而且能够做到以前的计算机无法做到的很多事。谁能够想到计算机能够在半秒内从全世界范围内找到符合你的查询的百万量级的文档？正如科幻作者Arthur C. Clarke说的，“任何足够先进的技术看起来都像具有魔力一般”。

然而，以上这些观察结果并不能安慰一些观察者。在2000年，Sun Microsystems的比尔·乔伊（Bill Joy）阐述了他对于一些危险的智能可能会出现、失控，并且挫败或压倒人类试图阻止它的措施的担心[2]。在2014年，物理学家斯蒂芬·霍金（Stephen Hawking）也发出了类似的警告，“对于人类而言，人工智能可能是最为糟糕的事物”[3]。谷歌公司建立了人工智能伦理委员会来监督其在人工智能方面的工作，从而避免这些工作在二十一世纪里导致人类的灭绝。我们认为没有计算原理能够排除这种可怕未来的发生，我们也同意设计人员应该对大规模网络化的计算机系统付出极大关注来应对可能出现的风险。

然而，计算机终将超过人类并不是一个确定的预言。在1997年，IBM的超级计算机在国际象棋领域战胜了加里·卡斯帕罗夫（Garry Kasparov）。几年后兴起的自由式国际象棋比赛中，棋手通过咨询笔记本电脑上的国际象棋程序能够轻松击败国际象棋超级计

算机。当人类与机器进行“合作”而非“对抗”，这种组合能够产生极大地超越计算机的智能。

架构和算法

目前有一个常见的观点，即所有关于计算的巨大进步都是源于算法。例如，谷歌的成功看起来像是依靠其巧妙的页面排序算法。或者数据库回滚依赖于其原子化的事务算法。或者在线交易事务能够确保安全是依靠大量的 RSA 密码算法。在每一个显著的计算机成就之后似乎都有一个精妙的算法。

因此，对很多人来说，算法分析和编程就是计算机科学的核心。

但对我们来说，这个结论似乎并不正确。事实也并不这么简单，让我们来看看摩尔定律。摩尔定律认为，一块计算机芯片上的晶体管数量每两年增加一倍。从一开始这就是工业界的趋势。这为我们带来了每十年近百倍的计算能力的增加。在工业界有一个经验规律，任何一项可以使得一个重要过程增速十倍的技术都是具有潜在扰乱性的。摩尔定律远远超 243
出了这个规律，并使计算行业处于一个纷乱不断的过程中，这是一个非常重要的现象。

但是，摩尔定律的进步并不是仅仅由于算法的作用。而是由于材料的设计，对物理的最新理解，在芯片上的电路架构，以及在光刻技术、电路模拟器、清洁化生产和其他有关方面的进步。

没有谷歌的全球数据仓库的基础设施，谷歌的页面排序算法将无法工作，这些基础设施能够在半秒内调动成千上万个程序来找到符合查询的结果。

那么量子计算又如何呢？物理实验室正在寻找方法来利用能够表示原子的量子叠加态的“量子比特”来进行计算。这些进步将来源于新的物理学而非算法。如果他们成功了，大部分的现代密码技术可能会变得过时。

那么多核芯片呢？为了从芯片中获得更多的计算能力，设计人员生产了包含多个处理器的芯片，这要求编程人员编写多线程算法来获得最快的速度。如果他们成功了，将不仅推进摩尔定律继续向前发展几代，而且会引发一场编程的革命。

这些仅仅是计算机架构本身发生改变的许多例子中的一部分。新的架构促成了更快的算法，使得一些原本不可行的算法变得可行，也启发了那些能够最好地利用架构的新算法。

所以对我们而言，计算机的架构和它们所运行的算法一样重要。这在计算原理中是非常明确的。很多原理是关于运行计算的系统的。如果把原理的范围限制为关于算法的原理，而忽略关于架构的原理，那么我们就无法给出一个关于计算的全面图景。

这就是为什么本书中有如此多的系统原理的原因。

经验思维

计算机科学的批评家传统上认为计算机科学主要是数学（例如算法分析）和工程（例如架构设计和软件开发）。他们之所以反对计算机科学被称为科学，一部分原因是由于他们没有看到计算机科学对于实验方法的努力追求，一部分原因在于计算机科学的研究对象
244 和信息处理似乎是人为而非自然的。

过去的二十年发生了巨大的变化。计算机科学家已经在很多领域采用了实证方法，包括软件原型，计算机安全，调试，架构的定量设计，预测网络的响应时间，人工智能，启发式算法的验证，以及其他多种方面。计算机科学家并没有放弃建模和分析，但他们已经在实验验证模型和分析中有了长足的进步。

此外，许多领域的科学家已经发现了一些自然发生的信息过程，并邀请计算机科学家来帮助他们了解这些过程。尽管计算机科学涉及的大都是人工（机器产生）的信息处理，但许多相同的原理也会用于自然的信息处理。计算领域中一个正在发展的领域就是与自然信息处理有关的。

计算机科学正在不断成长，和工程与数学一样，它也赢得了作为一门科学的声誉。

由于整个科学领域普遍的计算机运行和计算的影响，以及计算机领域和其他领域越来越频繁的交互，保罗·罗森布鲁姆（Paul Rosenbloom）认为计算机科学不仅仅是一个科学领域，而且还是一个堪比物理、生命以及社会科学的新领域[4]。

一个崭新的机器时代来临

正如18世纪80年代，蒸汽机成为自动化和强化手工劳动的转折点，现今，网络化的计算机已经成为自动化和强化认知工作的转折点[5]。这产生于本书所涵盖的所有事物的聚合：算法、系统和设计。算法产生了巧妙解决认知问题的新方法。系统和网络已经开发出了能够提供通用的运行这些方法的计算能力的范围和性能。而设计人员已经非常善于寻找算法和系统的新组合来为人类社会提供宝贵的贡献。

信息经济学和关于实体的经济学有着深刻的不同。物理的产品例如智能手机需要大量的前期开发成本和大量的销售来摊销这些成本。而电子产品现在可以以较低的成本被设计出来，并且以更低的成本传播——应用程序、应用程序开发工具、应用程序市场的爆炸
245 式增长见证了这一现象。数字对象的世界是丰富的而非稀缺的。应用程序是新经济的软件工具。应用程序的工作使得用户的生活变得异常轻松。你是一个徒步旅行者吗？利用GPS来定位你的旅途吧。希望阅读你的晨间报纸？装一个本地报纸的应用程序吧。想要快速地办理登记手续？装一个航空公司的应用程序吧。想要呼叫一辆出租车？装一个可以

在你所在区域使用的打车应用程序吧。智能手机和平板电脑本身并没有产生革命，应用程序把这些设备转化成了非常有用的实现认知工作的实用工具。

新的计算机时代最令人不安的一个问题是快速变化的就业市场。如果一个人的工作岗位是可以由机器自动执行的，那么他就有失去工作的风险。设计人员和软件工程师的工作岗位正在扩张，尽管他们的产品替代了那些可以自动化的工作岗位。我们的教育体系还没有跟上帮助失业者找到新的技能和工作的需求。教育者和政策制定者在这方面的工作显得不足。

我们的思维方式正在转变

想象一下在网络搜索中非常容易被找到的两张图片。一张是现代超级计算机的照片。例如，IBM 在阿尔贡实验室的蓝色基因超级计算机，在它的 72 个机柜中包含了 250 000 个处理器，每秒可以执行 10^{15} 次运算——是智能手机芯片速度的一百万倍。这种机器非常善于利用确定性算法处理大数据集。这种计算机没有智能。

另一张图片是一个互联网络图，这是一张极为漂亮的互联网网站之间数据连接的快照图，拥有更多入链的网站在图中显得更大更亮。互联网是一个人类和计算机无尽交互的有机体，在这个有机体中人和计算机互相放大彼此的能力。这个有机体不断改变着它的结构，有些改变甚至是有破坏性的。这个图景代表了来自十亿计的计算机和数十亿计的人所组成的超级计算机。这个有机体具有智能——能够收集、增强、协作参与其中的每一个成员的智能的能力。

互联网有机体并不会取代机器。这是一个建立在机器、移动设备、连接以及和人类的交互上的一个崭新的系统。计算机网络是这个有机系统的基础设施。

这两幅图片也代表了不同的理解世界的方法。机器的视角代表了科学的进步，它似乎知晓所有的数据，能够预见未来发生什么，并进行控制。有机体的视角展示了一个充斥着不确定性，不可预测的和混乱的世界，这个世界没有规则地不断发展。我们对设计和架 246
构的态度是在机器时代形成的。针对互联网有机体时代的新的设计和架构原理无疑会陆续出现。教育系统也将为人们在这个新兴的世界中生活做好准备，并且合理利用技术给子孙后代留下一个更美好的世界。

设计的核心性

综合起来看，这些思考把设计师以及他们的工作放到了计算机进步和创新的中心位置。机器的智能化并非来自于架构或算法，而是来自于设计师的工作。设计师在软件和机

器要为用户产出的含义上费尽心思。他们精心地进行设计，从而使得期望的含义能够如实地提供给用户。当编程人员是设计师的时候，他们具有最大的影响力，否则，他们只是实现别人设计的程序员。

为了强调这一点，我们对设计进行了10个章节的论述。有经验的设计人员使用设计原理，以引导他们完成可靠、踏实、实用、安全、保险的计算机系统。虽然这些原理不是自然规律，但它们在实现计算运行的过程中是非常重要的。

247 让我们为设计师鼓掌，为他们让我们能够完成我们的工作而鼓掌吧。

各章概要

第 1 章　作为科学的计算

计算是一门相对年轻的学科。其作为一个学术研究领域确立于 20 世纪 30 年代，确立的主要标志是由 Kurt Gödel（1934）、Alonzo Church（1936）、Emil Post（1936）、Alan Turing（1936）等人所发表的一组重要论文。这些研究者敏锐地意识到了自动计算（automatic computation）的重要性，并开始为“计算”这个概念建立坚实的数学基础。他们回答了“什么是计算”这个问题，并探讨了实现自动计算的不同模型。这个领域的第一个 40 年关注对计算技术和网络的持续发展和优化。从 20 世纪 80 年代开始，这个领域开始将注意力向外部转移，与计算科学以及其他许多领域产生了重要的交互关系。当意识到计算机自身只是研究信息过程的一种工具，这个领域开始将其关注点从计算机器转移到信息变换上。

第 2 章　计算领域

计算相关的实践活动来源于设计和使用计算系统的人。计算相关的实践者在长期的实践活动中形成了数量众多的实践区域，我们称之为计算领域。目前已经存在了几十种计算领域。同一计算领域中的人们共同面对相似的问题，分享相同的技巧和方法，并与其他计算领域中的人们进行相似的交互。他们享受计算的基本原理带来的权利，同时也受到这些原理的限制。我们介绍了 4 种具有重要现实意义的计算领域：信息安全、人工智能、云计算、大数据。对其中的每一种计算领域，我们对其演化历史进行了简要介绍，然后说明该领域中存在的不同角色、每种角色关注的问题、该领域涉及的基本计算原理以及从其他非计算领域中借鉴的基本原理。这种分析方法适用于对任何一种计算领域的分析。

第 3 章　信息

计算机的作用是存储、变换和传输信息，这种说法令很多人感到困惑，因为信息是抽象的，并且带有一定程度的主观性，而计算机只能处理确定的信号和事物状态。那么，什么是信息？是信号和状态吗？是人为确定的含义吗？事实上，信息是一个混合体，是由信号和事物状态组成的有含义的模式串（编码）。“有含义”这个词让人们注意到：模式串

就是由设计者设置的被编码的解释；机器通过设计者给出的规则来处理编码，设计者需要保证解释出来的结果能够符合预期。信息论是用来处理形成模式串的编码理论，它告诉我们：熵规定了一个可解密码的最少位数，我们可以在编码中增加足够多的冗余位来保证在噪声存在的情况下，信息也能被 100% 地可靠传输，我们也可以通过使用更短的编码表示信息，来对文件进行压缩。关于计算，一个明显的悖论是机器处理信息的时候不关心信息含义，而使用者又是通过与机器的交互来知道这些含义的。不过当我们意识到信息的含义来源于设计者的意图时，这个悖论便消失了。

第 4 章　机器

计算机是加工信息的工具，它执行一系列由电子电路实现的计算指令。1944 年问世的存储程序计算机实现了算术运算、逻辑运算、条件选择以及循环迭代的指令。如此简单的一个指令集，却具备强大的功能，我们可以通过编程来让计算机来执行任何可计算的功能。当然，这样的简便也是有代价的，即使比较直观的功能也可能需要上亿条指令来实现。不过，因为计算机执行速度如此之快，我们可以承受这种代价。在 20 世纪 50 年代需要数周才能完成的任务，如今在眨眼之间就可以完成。计算机编程语言使得程序员可以简短地描述他们想做的事情；编译器则在保持原始表达含义不变的条件下，将程序翻译成机器指令。和机器指令相比，编程语言增加了一些额外的概念，例如子程序调用和中断响应，这些需要专门的机器指令来支持。计算电路的设计者发现，由于不确定性原则，机器设计并不总是能得到好的结果——当电路必须在规定的时间内区分两个近乎同步的信号时，这可能导致计算机停止运行或崩溃。当时钟频率提高时，这种风险也会随之增大。我们可以通过降低时钟频率来减轻这种风险，或者使用自同步电路，持续等待直到可以将近乎同步的信号区分开来。

第 5 章　程序设计

程序员很早就意识到他们将会耗费很多的时间去发现自己编写的程序中存在的错误或运行程序的机器中存在的错误。他们发明了程序设计语言来应对这种错误问题。程序设计语言使得程序员能够通过具有精确语法格式的简单表达式来说明程序的行为；然后，编译器将程序转换为对应的机器指令序列（这个序列严格反映源程序的行为）。目前已经存在数千种的程序设计语言。每一种语言都有其适用领域。每一种语言都具有精确的语法格式和相应的编译器。计算机的能力可以通过虚拟机的方式得到扩展：通过虚拟机，可以在指令集合中增加新的复杂指令（被实现为子程序）。虚拟机使得 Java 语言可以运行在任何

一种计算系统中，使得一个操作系统的多个版本可以同时运行在一台硬件计算机上，使得操作系统和网络协议的功能层次结构得到优雅的实现。

第 6 章　计算

计算机要花费多长时间才能完成既定的任务呢？为了回答这个问题，我们计算了执行一个程序需要的指令数。每一个指令都需要一段时间去执行，不管计算机单独执行一个指令有多快，所有的指令加起来还是会有明显的延迟。算法可以根据它们的执行时间来进行分类：一些算法的执行时间与它们处理的数据大小成线性关系；一些成平方关系；一些成指数关系；还有一些算法比指数增长关系更差。我们关注的大部分都是那些执行时间成指数增长的算法，因为这些算法解决的问题一般都比较重要，并且如果数据集很大的话，这些算法本质上无法完成。即使宇宙里的每个颗粒都可以看作一个存储单元，这样大的内存也不足以解决这类大规模的问题。而且，即使我们拥有足够大的内存，解决这类问题所需要的时间会比太阳系的剩余寿命还长。我们研究了为什么计算机科学家认为能够利用快速算法解决的问题（“P”代表多项式时间）和只能用指数增长时间甚至更糟的算法解决的问题（“NP”代表非确定性多项式时间）是不同的。一般来说，现实中所有的问题都受到这个限制，比如在传输网络中分配资源或者在大数据集中找到最优子集等问题。在很多情况下，我们惊奇地发现一些启发式算法可以在合理时间内提供部分最佳答案（可以提供近似解）。最差的情况是任何计算机都不能解决某种问题，比如说程序中有死循环或者存在恶意病毒。这种限制是由信息自身的逻辑带来的。

第 7 章　存储

如果不能有效地存储和检索信息，我们就无法进行计算。信息存储和检索的四个主要原则是命名、映射、验证和定位。命名是指标明一个计算过程申请的所有条目的方法，这些方法包括记录地址、数据库查询以及非结构化数据的关键字检索。映射是指在名称和对象之间建立一个访问路径并且利用这种映射去传递信息。验证是指确认请求访问的用户是否拥有访问权限，从而拒绝未授权用户的访问。定位是指配置存储器层次结构或网络中的信息，从而使处理器访问的距离最小。一般情况下，由于 CPU 缓存和硬盘的访问时间差距很大，或者一个很忙的服务器队列很长的时候，定位是至关重要的。局部性原理是指在一段时间内计算引用的内存趋于聚集在一个较小的连续区域，这是所有高效定位方法的基础。局部性原理和计算本身的概念密切相关：同一个算法中的每一个操作都是对固定范围的数据结构进行读取或修改，所以计算是一定具有局部性的。

第 8 章　并行

在传统上，我们更注重计算中的串行算法，但现实世界中的大多数活动都是由偶尔同步的自治代理操作并行完成的。目前，并行处理是最普遍的计算模式。并行有两大类：协作并行和竞争并行。当多个进程一起，互相同步完成一个共同的目标时，就会形成协作并行。例如，利用 10 000 个处理器来加速天气预报。当很少或者没有同步关系的多个进程排队访问有限的网络资源时，则会形成竞争并行。本章探讨了协作并行中的一系列问题，包括避免竞态条件、对使用中的共享资源加锁、避免死锁等。在大型系统中，无法保证相互独立的各子任务的执行顺序。它们的顺序在每一次执行过程中都可能不同，使得其结果不可预测、其行为存在潜在的不安全。在这样的环境中，因为错误行为无法重现，调试几乎是不可能的。唯一的解决办法就是遵守设计协议，避免不安全的竞态条件和死锁发生。这些方法大都隐藏在操作系统内部，一般的程序员并不需要处理它们。然而，多核芯片技术迫使所有程序必须熟悉协作并行方法——这是程序员面临的一个重大转变。

第 9 章　排队

本章探讨了竞争并行的一系列问题。一个计算系统被建模为一组通过网络连接起来的服务器。当用户向系统提交作业（工作请求）后，作业从一个服务器移动到另一个服务器，收集各服务器上的服务结果，直到所有的服务器都结束。然而，在每台服务器上，作业都会排成队列，特别是当服务器的处理速度无法跟上请求的速度时。排队延迟可能严重影响系统对一个作业的响应时间。如何预测一个网络系统的响应时间和吞吐率？简单统计一个算法的步骤数仅仅是回答这个问题的开始。我们需要援引排队论中的原理来回答性能预测的问题，同时发现并消除系统中的瓶颈。计算机科学家发现了如何利用排队网络来精确建模大型计算系统的方法，并开发了用这些模型计算预测结果的快速算法。同样的模型在工业问题和业务工作流问题中也可以使用。当搜索引擎使用数千个处理器来快速处理一个查询时，它很好地展示了排队网络原理——并行消除了瓶颈。

第 10 章　设计

设计在计算领域的发展历程中始终是一个核心议题。计算领域第一代的设计者为我们留下了一个非常好的设计方案，即：存储程序计算机（又被称为冯·诺伊曼体系结构）。这种设计方案至今仍在使用。存储程序计算机使用指令集合作为用户接口，孕育了程序设计这一技术职业。它也导致了调试技术的出现，调试技术提供了一种系统性的方式去发现程序中的错误。软件设计者在实践中形成了 5 种重要的设计准则：需求、正确性、容错

性、时效性、适用性。需求关注于系统行为的准确描述；正确性关注于如何在构造过程中防止错误的发生；容错性关注于在错误被修正 / 去除之前最小化错误的后果；时效性关注于如何对系统进行配置从而保证计算结果的准时性；适用性关注于用户满意度。为了满足上述设计准则，软件设计者探索出 5 种有效的设计模式：层级式聚合（体现在抽象、分解、模块化、信息隐藏中）、封装、级别（层次）、虚拟机、对象。这些模式被实现在语言、应用程序、操作系统的结构中。设计对于决定一个计算系统的成败具有非常关键的作用。如何进行优秀的设计可能是计算领域中最大的一个挑战。

第 11 章　网络

Internet 是最重要的计算技术之一，该章是关于 Internet 的一个案例研究。Internet 的每一个主要部分都利用了第 1 章提到的 6 大原则。这些主要部分紧密地联系在一起，使得 Internet 变成了一个可靠、可信、可持续扩展的庞大系统。不同于电话网络中使用的线路转接技术，Internet 使用包交换技术来传递信息。在自动路由的过程中，当路由到网络中某些损坏部分时，节点和连接的丢失是不可避免的，包交换技术可以使得 Internet 在这种情况下也能正常工作。IP 协议对每个主机分配一个单独的整个 Internet 范围内可识别的地址，使得 Internet 跨过了数百万的局域网络。TCP 协议将数据的字节流分割成了若干个连续的有序号的包，从发送端发送，在接收端接收并重新组装成原始数据流。包的应答机制向发送者保证了包已经被接收，超时机制提醒发送者重新发送未被应答的包。DNS 给予了每个主机一个唯一的字符名字并将这些字符名字对应到其 IP 地址。服务器 – 客户端模型使用 TCP 协议建立联系并访问这些网络，一个国际化管理系统将基本的网络服务分配到固定的标准端口上。万维网（WWW）能将任意主机的文档连接到网络中的任意数字对象，并为连接的用户提供一份对象的拷贝。这个庞大的系统使用了包括通信（编码和错误修正），计算（安全加密、路由表），协调工作（协议、多路复用技术），复用（命名、地址、超高速缓存技术），评估（流量控制、通信量研究）和设计（网络层、服务器 – 客户端系统、端对端协议）等众多原理和技术。对网络中连接的研究引出了新的诸多网络连接模型和一个新的叫做“网络科学”的研究领域。

第 12 章　后记

对贯穿于本项目的一些关键问题的观察与思考，我们进行了总结：无思维机器，智能机，架构与算法，经验思维，新机器时代，转变我们的思维，以及设计的中心地位。

注　释

序

1. W. M. Kahan, "*How futile are mindless assessments of roundoff in floating-point computations: Why should we care? What should we do*? (Extract), " in Proceedings of the Householder Symposium XVI on Numerical Linear Algebra , p. 17, 2005.

前言

1. Brynjolfsson, Erik, and Andrew McAfee. 2014. *The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies.* W. W. Norton.

2. Rosenbloom, Paul. 2012. *On Computing: The Fourth Great Scientific Domain.* MIT Press.

3. 摩尔定律是经验观察，即计算机芯片的能力大约每两年就会翻一番，而其价格不变，进而可以得出计算能力每 10 年会增长数十倍。

4. Bacon, Dave. 2010. *Computation and fundamental physics*. http://ubiquity.acm.org/article.cfm?id=1920826.

5. Snyder, Lawrence. 2012. *Fluency with Information Technology: Skills, Concepts, and Capabilities,* 5th ed. Addison-Wesley. http://www.fluencywithinformationtechnology.org/.

6. http://www.csprinciples.org/.

7. Wing, Jeannette. 2006. *Computational thinking*. Communications of ACM 49(3):33-35. http://doi.acm.org/10.1145/1118178.1118215.

8. 延伸阅读书目如下：

- Eck, David. 1995. *The Most Complex Machine.* CRC Press.
- Biermann, Alan. 1997. *Great Ideas in Computer Science* (2nd ed.). MIT Press.
- Hillis, Danny. 1999. *The Pattern on the Stone: The Simple Ideas That Make Computers Work*. Basic Books.
- Harel, David. 2000. *Computers Ltd: What They Really Can't Do.* Oxford.
- Petzold, Charles. 2000. *Code: The Hidden Language of Computer Hardware and Software.* Microsoft Press.
- Berlinski, David. 2001. *The Advent of the Algorithm: The 300-Year Journey from Idea to the Computer.* Mariner Books.

• Witten, Ian, Marco Gori, and Teresa Numerico. 2006. *Web Dragons*. Morgan Kaufman.
• Abelson, Hal, Ken Ledeen, and Harry Lewis. 2008. *Blown to Bits: Your Life, Liberty, and Happiness After the Digital Explosion*. Addison-Wesley.
• Reed, David. 2010. *A Balanced Introduction to Computer Science*, 3rd ed. Addison-Wesley.
• Rushkoff, Douglas, and Leland Purvis. 2011. *Program or Be Programmed: Ten Commands for a Digital Age*. Soft Skull Press.
• Blum, Andrew. 2012. *Tubes: A Journey to the Center of the Internet*. Harper Collins Ecco.
• Brynjolfsson, Erik, and Andrew McAfee. 2012. *Race Against the Machine: How the Digital Revolution Is Accelerating Innovation, Driving Productivity, and Irreversibly Transforming Employment and the Economy*. Digital Frontier Press.
• Gleick, James. 2012. *The Information: A History, A Theory, A Flood*. Vintage.
• MacCormick, John. 2012. *Nine Algorithms That Changed the Future*. Princeton University Press.
• Fortnow, Lance. 2013. *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press.
• Brynjolfsson, Erik, and Andrew McAfee. 2014. *The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies*. Norton.

9. Denning, Peter. 2003. *Great Principles of Computing*. Communications of ACM 46 (11):15-20.

10. 我们的同事，已故的 Jim Gray，ACM 图灵奖获得者，曾大力支持我们对计算机科学中最深入原理的探究。他建议我们聚焦到“宇宙的原理”（cosmic principle）——那些在任何时间、在宇宙中任何地方都有效的原理——避免将好创意和这样的原理混淆在一起。例如，他认为虚拟内存是一个难以置信的好创意，而局部性则是一种深刻的原理。

11. Williams, Archibald. 1911. *How It Works: Dealing in Simple Language with Steam, Electricity, Light, Heat, Sound, Hydraulics, Optics, etc. and with Their Application to Apparatus in Common Use*. Thomas Nelson and Sons.（可免费下载。）

第 1 章

1. Matti Tedre（2014）对数学、工程、科学等领域的传统在计算领域中的体现进行了非常完整和详细的历史回顾。在 20 世纪 80 年代，来自不同领域的传统之间偶尔也出现了冲突。例如，当时有数学家认为，计算理论是真正的计算机科学，计算机工程只是一种技术；而有些计算机工程师则认为数学并没有为构造实际可运行的计算机和计算机网络提供任何帮助。他们甚至开始争吵软件工程是否应该被作为计算机科学的一部分，或者是否应该被作为一个单独的工程学科。随着计算领域逐渐对这些来自不同领域的传统进行了有

效的融合，这些零星出现的冲突基本上不复存在。在对众多其他领域提供服务的过程中，计算作为一个独立领域的地位得到确认。

2. 在本书中，我们将原来的 7 种类别简化为 6 种。被去除的分类是自动化：自动化是计算的一个高层领域，关注于如何以及何时对人类的认知任务进行自动化。在第 2 章中，我们将该领域称为“人工智能”。

第 3 章

1. James Carse (1986) 在 1986 年对游戏也做了相似的区分。他说“现在至少有两种不同的游戏。一种叫做有限游戏，一种叫做无限游戏。玩有限游戏是为了能够获得最后的胜利，玩无限游戏的目的是为了能一直玩下去”。这两种游戏有着本质上的不同。尽管他不是在讨论计算机，这种区分方式也进一步深刻体现了有限算法和无限交互系统的不同。

2. 发送者可以以一次一密的方式加密信息，方法是首先生成一个与信息 bit 数一样的随机串，然后将两个串逐位进行异或操作，得到的结果就是信息的密文（异或操作是当两位相同为 0，不同为 1）。接收者将密文重新与随机串进行异或来得到明文。香农证明了密文的熵是最大的，这意味着窃听者得到密文并不会得到任何有用的信息。那明文和其中的信息去哪了呢？其实信息并不在秘钥或者密文之中，而是在由设计者定义的明密文转化之中。这对加密 - 解密的方式很好地保持了原文的含义。一些其他的加密系统使用更短的秘钥，在这种情况下，密文中一些冗余的信息可能会被破译者所利用。

第 4 章

1. Hennessey 和 Patterson 的书（2011）完美地覆盖了计算机体系结构的各个方面。我们通常将存储程序式计算机的原始结构归功于 John von Neumann（1945），但是从他发表的与 Eckert、Mauchly、Burks 和 Goldstine 等人的会议记录来看，大部分体系结构的想法都来自于 Eckert 和 Mauchly，而不是他本人。

2. 1956 年，IBM 发明了新的硬盘存储系统 RAMAC（Random Access Memory Accounting Machine，随机存取磁盘驱动器），这是第一个能够随机地进行存储访问的硬盘。随机意味着完成一次访问的时间是由寻道时间（读写头定位）和延迟时间（旋转定位）组成的一个随机变量。现在 RAM 指的是计算机芯片中的主存，但是随机还有另外一个意思，那就是随机选中一个地址，对它的访问时间都是固定的。

3. Corrado Bohm 和 Giuseppe Jacopini（1966）证明了任何可计算的功能都可以通过只由这三种结构组成的程序进行计算。这个理论被证明是正确的，使得程序更容易理解，

同时被当作“结构化程序设计”的基础。几年之后 David Harel（1980）将这种理论追溯到冯·诺依曼体系结构本身的设计和在 20 世纪 30 年代由 Stephen Kleene 证明过的标准化定理。

4. Jan Lukasiewicz（1957）表示他第一次看见符号这个概念是在 1924 年。Arthur Burks、Don Warren 和 Jesse Wright（1954）是第一次注意到逆波兰表示法可以简化表达式的机械计算。Fritz Bauer 和 Edsger Dijkstra 是在 20 世纪 60 年代初各自发现了这点（维基百科）。

5. 递归可以使程序变得更简单。例如，我们可以将一个排序程序写成下列形式：SORT (list) = {SORT(left half of list)；SORT(right half of list)；MERGE (left half, right half)}，边界条件是 SORT(empty list) = empty list。每一个内层调用 SORT 函数的输入列表必须比外层调用的要小。

6. 这一准则与量子物理学中海森堡测不准原理非常相似。测不准原理是说位置和动量的标准差的乘积至少为 10^{-34} 焦秒。其中一个量越确定，另一个量的不确定程度就越大。海森堡原理的部分原因是，观测行为本身就会增加或减少被观测粒子的能量，但这只会发生在原子尺度的电子上，而不会在宏观尺度的电线电流中。选择不确定原则并不是海森堡原理的一个实例。

7. 异步电路（见第 8 章，并行）是由一些用就绪 - 确认信号进行交互的模块组成。它们这样设计是为了在介稳状态下不会产生就绪和确认信号。异步电路不需要时钟，通常比时钟电路要快，因为模块就绪后就可以发射信号，不需要等到下一个时钟周期。

第 5 章

1. 关于程序设计语言的更一般介绍可以参见 Pierce (2002) 和 Louden (2011)。

2. 通过 Google 搜索引擎，我们发现来自澳大利亚佩思 Murdock 大学的一个小组声称已经建立了一个包含 8500 种程序设计语言的数据库。不过我们没有验证这一数据是否真实。

3. 这些数据可以在维基百科或其他一些来源中看到。一篇发表于《Harvard Business Review》的文章（Flyvbjerg 和 Budzier 2011）指出：IT 领域的平均预算超支率在 27% 左右；六分之一的项目预算超支两倍以上，开发日程延期 70% 以上。

4. 将高级语言编写的源程序自动翻译为机器代码的思想可以追溯到 20 世纪 50 年代。经过多年的研究之后，人们才学会如何解析程序并生成高效的机器代码。今天，自动翻译已经成为一个子领域。Aho、Lam、Sethi 和 Ullman (2006) 的图书对这个子领域的理论与实践应用进行了非常全面的论述。本书对这些理论进行提炼，仅保留了最本质的内容。

Brown 等人（2012）的图书讨论了两个著名的 Unix 程序：Lex 和 Yacc。这两个程序支持基于程序语言语法的 BNF 范式生成编译器。

5. 这种情况大多出现在那些在多种平台上运行的标准化程序。例如，HTML、Java、Javascript 语言具有明确的国际标准；但是，web 设计者仍然需要将其编写的页面在多种不同的平台上进行测试，因为在不同的平台上，一段相同的程序往往会体现出不完全相同的行为。

第 6 章

1. $O(n^3)$ 复杂度对于矩阵乘法来说不是最优的。Volker Strassen (1969) 发现了一种方法使得复杂度降到了 $O(n^{2.807})$，之后 Coppersmith 和 Winograd (1990) 发现了另一种方法使得复杂度降到了 $O(n^{2.37})$，但是这些快速算法非常复杂并且不那么直观。

2. 早在 1897 年，人们就对背包问题的各种变种有所研究。然而，直到 1930 年，数学家 Tobias Dantzig 才在他的一篇文章《数：科学的语言》中给其命名。

3. 参见 http://www.math.uwaterloo.ca/tsp/sweden/ 和 http://chern.ie.nthu.edu.tw/gen/ 12.pdf.

第 7 章

1. 映射功能是由 CPU 中一个叫做内存映射单元（MMU）的硬件实现的。MMU 包含一个叫做转换后备缓冲器（TLB）的小的高速缓存，它保存了最近的从访问页面到页框的映射。如果要访问的目标页正好在 TLB 的列表上，MMU 可以跳过页面查找，从而节省一次内存访问。假设忽略访问页面表的延迟，TLB 可以使映射过程提速到只用原来 1% ~ 3% 的时间。

2. 下面是关于能力系统的文献的一些例子。Jack Dennis 和 Earl Van Horn（1966）提出了能力系统这个想法。Robert Fabry (1974) 发现能力寻址是解决数据共享问题的最好办法。Bill Wulf 和他的同事（1974）实现了链接到对象的能力内核。Maurice Wilkes 和 Roger Needham (1979) 在剑桥大学建立了一个基于能力的机器和操作系统。Henry Levy (1984) 写了一个关于能力系统及其工作原理的总结和概述。Mark Miller (2003) 反驳了几个关于能力系统的错误看法。

3. Tahoe Least Authority File System (https://tahoe-lafs.org) 是一个使用能力寻址的开源文件系统。每个服务器都不能越权访问，因为它们只能获得恰好能完成它们任务所需的能力。如果某些对象发生了错误，也不会扩散到其他的对象。

4. MIN 策略是由 Belady（1966）为虚拟内存提出的最优策略。如果一个页面缺失，

新载入的页面就会替换一个下次访问时间最远的已载入页面。不管内存大小，没有其他的固定分区存储策略会产生比 MIN 策略更低的页缺失。然而，MIN 策略不能改变 RAM 的分配，它不能移除已经加载的页面，除非有其他的页面需要被加载进来。相比之下，VMIN 策略的做法是，如果可以预见到在阈值时间内这个页面不会再次被使用，就立刻删除这个页面。当我们调整 VMIN 的阈值时间，使得它内存分配平均大小等于 MIN 策略下固定的内存分配大小，VMIN 策略产生的页缺失会更少。

5. IBM 研究中心的 Les Belady 和麻省理工学院的 Peter Denning 在 1966 年都提出了局部性原理，并共同对其进行了研究证明。Belady (1966) 用它解释了分页算法的非随机性能，Denning (1968a) 描述了程序的内在内存需求。从那之后很多研究人员开始在许多系统中研究这一原理。局部性原理是计算机科学中被广泛证明的原理之一（Denning 1980），现在仍然被用来研究缓存性能的最优化问题（Xiang 等，2013）。

6. 这应该按程序的虚拟运行时间来度量，一次内存访问计为单位时间长度。通常忽略由中断（比如磁盘请求）造成的延迟，但是如果这些延迟与性能分析相关，就会被计算在内（Denning 1980）。

7. 抖动是指当多任务负载超出一个动态变化的临界值时系统吞吐量的突然崩溃现象。当 RAM 太小，无法容纳所有活动程序的工作集时，这种情况就会发生。程序如果空间不足就会窃取其他工作集的页面，造成更多的页缺失。很快所有程序就会全部失去工作集，然后在磁盘中排起队列等待系统解决它们的页缺失问题（Denning 1968b）。排队网络模型显示，当不断增加的页面需求使得分页磁盘成为整个系统的瓶颈时，抖动现象就会发生（Denning 等，1976）。

第 8 章

1. 真正的天气预测要复杂得多。它会采用六个面相接触的三维立方体，并且还要考虑仅在角上接触的相邻立方体产生的影响。气压公式会更复杂，要考虑风速和风向的影响。

2. Modula 是 Pascal 的一个修订版，1977 年到 1985 年间由 Niklaus Wirth 设计开发。Smalltalk 是由 Xerox PARC 的 Adele Goldberg（1983）、Alan Kay、Dan Ingalls 等人于 1980 年创造，并于 1988 年成为 ANSI 标准。CLU 是由 MIT 的 Barbara Liskov(1977) 及其学生开发的。实现了 Hoare 的 CSP 模型（1985) 的 Occam 是由 INMOS 公司的 David May 于 1984 年开发的，INMOS 公司是一家生产用于并行超级计算机的被称为 transputers 芯片的制造商。

3. MIT 的 Jerry Saltzer(1965) 把进程定义为“在处理器上执行的程序”。MIT 的 Jack

Dennis 和 Earl Van Horn(1966) 把进程定义为“一个指令序列的控制轨迹”，而贝尔实验室的 Vic Vyssotsky 则把进程定义为“线程”。埃因霍温技术学院（Technische Hogeschool Eindhoven）的 Edsger Dijkstra(1968a，1968b) 将进程定义为“随着程序中指令执行的 CPU 状态序列”。

4. 在 CSP 中，Hoare 将 Dijkstra 的信号量替换为一种更简单的协作机制——会合（rendezvous），即要交换数据的两个进程达到各自的会合点，此时数据通过一个通道在两者之间交换。

5. 人们通常认为是 Nico Habermann(1969) 为死锁建立了第一个形式化模型。他指出操作系统可能会进入到目前没有死锁但未来肯定会死锁的不安全状态。他证明了“银行家算法”可以确定一个给定的资源分配是否安全。Coffman 和 Denning(1973) 讨论了其他一些关于安全性的算法，由于开销的原因，这些算法很少会用到。

6. 为了形象地说明这一点，假定满足了约束之后，仍然有一组死锁的进程。死锁进程所持有的编号最大的锁不会被释放，因为有一个死锁进程持有它。该锁的持有者本身必定也在等待一个锁，按照约束，这个等待的锁的编号必须更大。这与死锁存在的假设相矛盾。

7. Dijkstra 的著名的哲学家就餐问题可以用该方法来解决。设想一张圆桌周围有五个座位，每个座位前摆着一个盘子，每两个盘子之间有一把叉子。经常有哲学家过来坐下，从盘子中间的碗里夹面条吃。要吃到面条，哲学家需要从他的盘子两边拿到两把叉子。当他们都试图同时拿起同一侧的叉子（比如右边），就可能会产生死锁：现在，每个人都在等待他的邻居放下叉子。如果每把叉子都编号了，并且每个哲学家都先拿起编号较小的叉子，这个循环等待就不会发生。

8. Richard Karp 和 Raymond Miller(1966，1969) 首次证明了这一定理。Coffman 和 Denning(1973) 则在操作系统环境中证明了该定理。Brinch Hansen(1973) 宣布它是操作系统的一个重要原理。

第 9 章

1. 对应用到计算机系统和网络中的全面的排队论理论和方法感兴趣的读者，我们推荐下列书籍：Kleinrock (1975，1976)、Kobayahsi 和 Mark (2008)、Lazowska 等 (1984)、Menascé 和 Almeida (2002)、Menascé 等 (1994) 和 Stewart(2009)。

2. 利用率法则和利特尔法则与稳定状态下随机排队系统中著名的极限定理（limit theorem）非常相似。极限定理经常在实际测量中被验证，不是因为达到了稳定状态，而是因为测量的各个量服从运算法则 (Buzen 1976)。

3. 指定一系列单元，称为 $H[n]$，$n = 0, 1, \cdots, N$。这些单元分别保存 $p(n)$ 的试验值，假设 $p(0)=1$（这个假设并不正确），在各单元中插入平衡方程，作为从 $H[n-1]$ 计算 $H[n]$ 的公式。这样得到的各试验值服从平衡方程，但其和不等于 1。我们可以使用如下方法让它们和为 1：首先，创建一个新的单元“sum”表示 $H[0] + ... + H[N]$；然后，创建一组新的单元 $p[n]$，$n = 0, 1, \cdots, N$；最后，在 p 单元中放入公式 $p[n] = H[n]/\text{sum}$。这样，p 单元中各值的和为 1，且服从平衡方程。

4. 状态是一个向量 $(n_1, \ldots, n_K)$，其中 n_i 是服务器 i 的状态，所有 n_i 的和是系统的负载 N。我们可以用 N 个 1 和 $K - 1$ 个 0 组成的字符串来表示一个向量。由 0 所分隔的那些由 1 组成的子串（两个连续的 0 表示中间有一个长度为 0 的 1 串），其长度等于某个分量。所有可能的字符串的数量是 $(N + K - 1)!/ N!(K - 1)!$

第 10 章

1. 据说 Ada Lovelace 是人类历史上第一个数字计算机程序员；在 19 世纪 40 年代初期，她协助 Charles Babbage 进行分析机的设计。Babbage 没有能够完成分析机设计，Lovelace 设计的程序也从来没有真正运行过。从此以后，直到 20 世纪 40 年代初（100 年以后），才又有人开始从事程序设计的工作；直到这个时候，程序设计才成为一种技术职业。程序设计的实践产生了很多新的设计者，他们设计出了程序语言、程序编辑器、翻译器、调试器、版本控制系统、图形用户界面、应用程序等许多新的工具或功能。

2. Christopher Alexander 开启了建筑体系结构学派，被认为是“建筑的永恒之道”（1979）。他认为，有经验的设计者在进行设计决策时往往只依赖于一小组固定的模式。

3. 德国工业设计师 Dieter Rams 总结形成了优秀设计的 10 个方面，人们通常将其称为“Rams 设计原理”（来源：维基百科）。

4. 2007 年，模型检查的提出者 Edmund Clark、E. Allen Emerson、Joseph Sifakis 因为他们的杰出成就获得 ACM 图灵奖。

5. 错误限制经常出现在其他领域中。船只被划分为不同的区域；每个区域由水密封门相互隔离；这样，在船体发生局部漏水的情况下，船只也很难沉没。热气球被划分为不同的区域，以减少气球被刺穿后的风险。桥梁由数千个三角部件组合形成；这样，一个三角部件被破坏后，桥梁也很难倒塌。

6. 技术哲学家 Carl Mitcham（1994）将“技术知识”描述为：（1）制作的技巧（“know-how”）；（2）描述性规律（特定条件发生时应该采取什么行动）；（3）技术箴言（经验规则）；（4）技术理论（将科学理论应用至实践中）。在我们的术语体系中，基本原理是

对制作技巧的描述，模式是对描述性规律的描述，示意是对技术箴言的描述。

7. http://en.wikipedia.org/wiki/Software_design_pattern。在 2014 年，这个页面中列出了 9 个创建型模式、9 个结构型模式、15 个行为型模式、15 个并发型模式。

8. Xerox Palo Alto 研究中心的 Jay Israel、James Mitchell、Howard Sturgis（1978）提出了客户端 - 服务器模型。Alfred Spector（1982）、Andrew Birrell 和 Bruce Nelson（1984）基于这种模型实现了远程过程调用（RPC）。1984 年，Robert Sheifler 和 James Gettys（1984）在 MIT 开发出 X-Window 系统（Scheifler 等，1988）。X-Window 是一个一般性的客户端 - 服务器宿主系统：用户提供客户端和服务器代码，X-Window 则通过网络提供通信服务。

9. 设计者的才能和技巧不是一些琐碎的点。在互联网上可以很容易地搜索到很多的相关研究工作。其中的一个基本发现是，最优秀的程序员在生产效率上是入门级程序员的 10 倍或以上。优秀的程序员能在所有的细节层次上对一个大型系统进行想象，并将他们的想象快速变换为可实际运行的代码。对一个软件公司而言，寻找到一个具有 10 倍工作效率的程序员并付给他 / 她双倍的薪水，将是一个非常划算的买卖。这远比雇佣 10 个入门级程序员并尝试对他们进行管理要划算得多。

第 11 章

1. 网络工程师对网络和主机进行了区分，网络是一系列传递数据包的路由器与链接的集合，主机是通过标准接口接入网络的系统。最初的 ARPANET（20 世纪 70 年代）是一个称作“接口报文处理器”（IMP）的数据包开关的网络。他们建立了很多所谓的子网络以便主机接入。IMP 实现了真正的网络功能，主机表现为消息的接收端或者来源。第一个版本的 Internet 软件（20 世纪 80 年代）使用叫做“网关”的机器将各个包交换网络互相连接。最终，网关被重新命名为路由器，主机通过路由器连接到网络。

2. 下述是关于 Internet 历史的一个简单介绍。关于这段历史的更多细节可以在 Barry Leiner(1996) 等人写的文档中，或者在 Katie Hafner 和 Matthew Lyon(1999) 写的书中找到。

MIT 的 J. C. R. Licklider 在 1960 年描述了极具想象力的未来场景：一个能够将全世界电脑连接起来的网络。这个网络能够支持所有的资源交换，拥有无所不在的计算能力、智慧端口、新的研究方法和新的商业类型（Licklider 1962，1963）。在 1961 年，UCLA 的 Leonard Kleinrock 设计了一种随机的通信网络模型，该模型能够将离散的信息向终点传送并且在中间网点进行排队（Kleinrock1961，1964）。RAND 公司的 Paul Baran 在 1964 年发表了一系列关于新的网络架构即分布式通信系统的论文，这一系统能够让连接和节点免于损坏 (Baran，1964a，1964b)，其中数字语音比特流被分成能够在路由器中传

送并且发送到目的地的小数据块。英国国家物理实验室的 Donald Davies 开始使用“包”（packet）这一术语去描述那些小的数据块，并且这一术语最终成为标准。

1967 年 DARPA 的 Bob Taylor 请 Larry Roberts 去领导一个新的项目，这一项目是建立一个能够实现那些想象场景（Roberts 1967，Roberts 和 Merrill 1966）的分布式的网络 ARPANET。Roberts、Len Kleinrock 和 Wesley Clark 共同合作进行这一项目。Clark 提议网络能够配置成由典型的小型计算机（接口报文处理器，IMP）组成的子网，它能传送数据包并以接口的形式为异质的计算机服务。当时还没人去构建 Baran 的想法。ARPANET 的前两个节点在 1969 年底的时候开始工作。

从 1970 年到 1983 年，Steve Crocker 领导开发网络控制协议（NCP）——ARPANET 中主机 – 主机的协议。作为网络控制协议的伴随物，文件传输协议（FTP）、远程登录协议（TELNET）和电子邮件协议（SMTP）被开发出来。在 1973 年，Vinton Cerf 和 Robert Kahn 提出传输控制协议（TCP）并最终成为 TCP/IP 协议套件。他们也设计了一种用于连接两个不同网络的地址结构和网关结构。以他们在 ARPA 的地位，设计师远见卓识地预见了因特网的进化和发展，虽然直到 1983 年 1 月，正式的因特网雏形才出现。TCP 用一种统一的方法将所有的子网连接成网络的网络（因特网）。TCP 为因特网提供了可靠的、具有效率的文件和信息传输。当 TCP/IP 协议套件在 1983 年成为标准协议时，NCP 不再被使用了。1973 到 1983 年期间是各种新协议得到实验性发展的一个时期。到 1981 年，ARPANET 已经标准化了因特网的一系列基本协议（RFCs 791, 792, 783）：寻址和基础的包交换使用 IP 协议，对于有序数据的传输使用 TCP 协议，对于数据报的传输使用 UDP 协议，对于主机间文件的手动传输使用 FTP 协议，远程登录使用 TELNET 协议，邮件传输使用 SMTP 协议等。

英国和法国政府也同样赞助了网络的早期研究。在 1967 年，英国的 Donald Davies 建立了一个单节点的包交换和模拟数据包网络。在 1972 年 Louis Pouzin 创建了一个叫做 CYCLADES 的网络，在这个网络中他把传递数据的包叫做数据报（datagram）(Pouzin 1973, 1974)。数据报的想法也被 TCP/IP 协议所吸收。然而，法国政府更希望在网络数据传输中保留类似于电话网络的结构。因此政府转而支持 X.25 协议的研究并使得 CYCLADES 项目研究终止。X.25 协议在美国鲜有人知，因为只有 GTE Telenet 公司提供此服务。X.25 协议的主要发起者是法国 CCETT 的成员 Rémi Després 和 Paul Gulnaudeau、加拿大 TCTS 的 David Horton 和 Anton Rybczynsky、美国 Telenet 的 Larry Roberts 和 Barry Wessler、英国邮政局的 Philip Kelly 和 John Wedlake、日本 NTT 的 Masao Kato。他们在 1976 年提出了 CCITT 采用的 X.25 协议标准。英国邮政局的 Chris Bloomfield 和法

国 CCETT 的 Bernard Jamet 也在之后的 X.3, X.28, X.29 协议的字符模式接口推荐中做出了短暂贡献。更多的细节可以参看 Després(2010)。

在 1981 年，美国国家科学基金 NSF 通过赞助 CSNET 进入到网络领域，CSNET 是一个连接全世界 CS（Computer Science，计算机科学）的研究部门和实验室的网络。CSNET 吸引了由 Larry Landweber、David Farber、Tony Hearn 和 Peter Denning 主导的高校联盟领导下的很多人员。CSNET 开发了 ARPANET TCP/IP 协议的系列版本，并且证明了其远优于电话的拨号连接和 X.25 连接，也没有使用 ARPANET 的租借电话线路的标准。在 1986 年，NSF 建立了骨干网络 NSFNET 来连接 NSF 的超级计算中心，这加强了其在网络领域的话语权。他们将本地网络连接到主干网络中并且向有商业往来的机构开放。到 1990 年，这个新兴的因特网已经拥有了 15 万的主机，并以每年一倍的速度增长。

在美国网络发展的同时，ISO 组织基于欧洲 X.25 协议也设计了 OSI 协议组。最初的 OSI 协议参考模型论文出现在 1978 年（见 Zimmerman 1980），协议标准于 1984 年由 ISO 公布。在接下来的十五年中，人们一直在争论 TCP/IP 协议和 OSI 协议到底哪个更适用于互联网。直到大约 1993 年，美国政府通过其国家标准技术局（NIST）认为 TCP/IP 是 OSI 协议的一个合理替代协议，TCP/IP 协议最终成为了因特网标准。

然后，Internet 开始缓慢地演化为商业事务的媒介。在 20 世纪 70 年代中期，ARPA 鼓励 IBM，DEC，和 HP 加入到研究项目中来。在 1985 年，ARPA 和 NSF 支持 CSNET 的成员在 ARPANET 上进行通信，但是当时这些成员都没有利用 CSNET 的连接来进行商业运作的想法。在 1989 年，NSF 首次将 NSFNET 的基础功能——Internet Service Providers(ISP) 交给商业代理商代理，代理商有 UUNET、PSINET 和 CERFNET，还有一些邮件功能代理商包括 MCI Mail、Compuserve、OnTyme、Telemail 和 GENIE。在 1989 年，由计算机科学家 Tim Berners-Lee 领导的 CERN 实验室在瑞士的一个小组，开发出了万维网，这使得在 Internet 上共享文档变得十分容易。随着第一个万维网的可视化接口——伊利诺伊大学 Marc Adnreessen 的 Mosaic 浏览器的发布，万维网在 1993 年变得非常热门。从那以后，各种各样的商业公司开始开发网页并通过互联网进行商业运作。

关于万维网的起源可以追溯到 20 世纪 60 年代 Ted Nelson 关于共享网络中数字出版的建议（Nelson 1980）。在他的设想里，作者可以通过一个称为 Xanadu 的系统来制作电子文档，Xanadu 系统可以自动处理出版、分发、版税和版权；作者可以使用超文本进行创作而不是线性地书写。Nelson 的想法一直没有实现，直到 20 世纪 80 年代，AutoDesk 买下了他的 Xanadu 公司，并且将这个软件公之于世。

同样在 20 世纪 60 年代，SRI 的 Douglas Engelbart 开展了一个通过合作来增加人类

智慧的项目。他的 NLS（在线系统）支持超文本对象的组织、可视化的文字交互、文档的协作管理、内嵌视频、弦键盘和鼠标控制。他的演示说明了 NLS 能够扩展到网络中，使得网络能够提升人类智慧。

在 20 世纪 80 年代，美国国家标准协会 ANSI 成立了一个关于计算机文档语言的委员会。传统的出版业将作者手稿用特殊符号标记以便印刷工知道如何设置印刷参数。在 1985 年 ANSI 提出了一个标准通用标识语言（SGML），一种描述个体标记语言语法的超语言。通过 SGML，作者、编辑和出版方能够自动地处理和修改文档。Tim Berners-Lee 是 SGML 的用户之一，他使用 SGML 的原理来定义关于网页的标记语言 HTML。他对 Internet 中的数字对象引入了 URL（Uniform Resource Locator）的标签，并且定义了 HTTP 协议来自动从 URL 连接中获取特定的文档。这结合了浏览器、HTML、HTTP、URL 和网络服务器的实体就是万维网的雏形。Berners-Lee 在 1985 年建立了万维网络联盟（W3C）来监督万维网有序地发展。

3. Kleinrock 和 Baran 提出信息和信息块的概念。术语“包”由 Donald Davies 在 1967 年提出并很快被公众接受。

4. 包的大小到底应该一样还是可以不同，一直是一个广为争论的问题。一样大小的包受到碎片的限制，即最后一个包会有无用的一部分数据，在最后一个包中，平均有超过半数的部分是无用的数据。大小变化的包可以很好地适应文件大小的改变。协议工程师需要衡量碎片的消耗和额外头说明的消耗。但是由于这个消耗取决于网络、文件分布和通信量大小，因此对于这个问题仍没有很好的答案。

5. 在网络术语中最常用的 32 位地址版本是“IP version 4”，即 IPv4。改进的使用 128 位地址的协议叫“IP version 6”，即 IPv6。到 2014 年仍有很多网络管理员没有升级网络来支持 IPv6。

6. 一篇在维基百科上的条目“TCP 和 UDP 的端口号列表”列举了上百个在 0 ~ 1023 之间的“常用端口”。大量使用 1024-49151 端口号的其他服务也已经被 IANA 注册了。

7. 电脑中的网络控制模块有一部分存储了 DNS 服务器的 IP 地址，DNS 服务器可以将电脑用户中的域名转化成 IP 地址。TCP 会使用 DNS 服务器。

8. 域名和 IP 地址的管理是非常复杂的，并且构建域名和 IP 地址的层次也十分困难。互联网名称与数字地址分配机构（ICANN，icann.org）主要负责决定顶级域名，并且为层次体系中下层域名选择管理策略。包含了顶级域名数据库的根服务器在进行更新时会使用一系列复杂但是安全的过程来完成更新。ICANN 会将计划的更新发送给国家技术信息局（NTIA），NTIA 将这些更新发送给 Verisign 进行确认；Verisign 生成新的根区域并且将

这些更新的拷贝发送给 13 个根区域的执行方，执行方包含了 ICANN（一个根服务器）和 Verisign（两个根服务器）。每个根服务器会在多个地点实现更新，以防止失败或者遭受攻击，并且这样能为全世界其余部分的电脑提供快速反应。在 2014 年，全世界已经有了 385 个根服务器（http://www.iana.org/domains/root/servers）。ICANN 是一个在加利福尼亚的全球性的非营利组织，在新加坡、伊斯坦布尔和洛杉矶都拥有国际董事会和执行机构。ICANN 的作用之一就是分配 IP 地址给区域互联网注册管理机构（RIR），这一功能由互联网号码分配局（IANA，iana.org）管理。RIR 制定了一系列由 ICANN 使用并由 IANA 管理的全球性规则。互联网工程任务组（IETF，ietf.org）设置了技术性的标准，这些标准规定了 IP 地址的结构、分配和布置。IETF 向华盛顿特区的非盈利组织国际互联网协会负责，并且在全世界都有办公室和分会。

9. 我们在第 10 章讨论了软件工程界中也非常有名的*层次原理*。层次是向下层提供功能的软件策略。在网络中，数据流在各层之间以包的形式上下流动，每一层只对包进行特定操作。在软件工程中，数据流是过程调用的参数，只允许向下层流动，并且只能通过过程返回的方式回到上层。

10. 2014 年，有人在 TLS 的 OpenSSL 实现中发现了一个缺陷。这个所谓的 Heartbleed 漏洞影响了 17% 通过 https 进行连接的主机，并且使得入侵者可以偷取密码和加密密钥。虽然该漏洞的补丁很快就发布了，但是依然有大量受到影响的网站不得不更换密码。所以尽管有仔细认真的设计，软件中依然会有缺陷！

第 12 章

1. 在《What Technology Wants》（Kelly 和 Kevin，2010，Viking）这本书中，有一个相关的例子。

2. Joy, Bill. 2000. *Why the future doesn't need us*. Wired magazine, issue 8.04 (April). http://archive.wired.com/wired/archive/8.04/joy.html.

3. 在 2014 年 5 月，史蒂芬·霍金对一部新电影《Transcendence》进行了关于操作系统发展智能和威胁人类的讨论。他不太确信计算机专家正在尝试控制随着资金增长的风险。他相信自主军事技术非常容易失控。

4. Rosenbloom, Paul. 2012. *On Computing: The Fourth Great Scientific Domain*. MIT Press.

5. Brynjolfsson, Erik, and Andrew McAfee. 2014. *The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies*. W. W. Norton.

参考文献

Abrams, Robert. 2011. *Encyclopedia of Computer Sciences*. Nova Science Publishers.

ACM Education Board. 2001. Curriculum 2001 recommendations. http://www.acm.org/education/curric_vols/cc2001.pdf

ACM Education Board. 2013. Curriculum 2013 recommendations. http://www.acm.org/education/CS2013-final-report.pdf

Aho, Alfred, Peter Denning, and Jeffrey Ullman. 1971. Principles of optimal page replacement. *Journal of the ACM* 18 (1):80–93.

Aho, Alfred, Monica Lam, Ravi Sethi, and Jeffrey Ullman. 2006. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Alexander, Christopher. 1979. *The Timeless Way of Building*. Oxford University Press.

Alfke, Peter. 2005. Metastable recovery in Virtex-II Pro FPGAs. Technical Report xapp094 (February), available from Xilinx.com website.

Arden, Bruce W. (ed.) 1983. *What Can Be Automated: Computer Science and Engineering Research Study (COSERS)*. MIT Press.

Bachman, Charles. 1973. The programmer as navigator. *Communications of the ACM* 16 (11):653–658.

Backus, John. 1959. The syntax and semantics of the proposed international algebraic languages of the Zurich ACM-GAMM Conference. *Proceedings of the International Conference on Information Processing*, pp. 125–132. UNESCO.

Bacon, Dave, and Wim van Dam. 2010. Recent progress in quantum algorithms. *Communications of the ACM* 53 (2):84–93.

Baltimore, David. 2001. How biology became an information science. In *The Invisible Future*, ed. P. Denning, 43–56. McGraw-Hill.

Barabasi, Albert-Laszlo. 2002. *Linked: The New Science of Networks*. Perseus.

Baran, Paul. 1964a. *On distributed communications: Introduction to distributed communications networks*. Rand Corp memorandum RM-3420-PR, available from http://rand.org/pubs/research_memoranda/RM3420.html

Baran, Paul. 1964b. On distributed communications networks. *IEEE Transactions on Communications Systems* 12(1).

Bard, Y. 1979. Some extensions to multiclass queueing network analysis. *Proceedings of the Fourth International Symposium on Computer Performance Modeling, Measurement, and Evaluation* (H. Beilner and E. Gelenbe, eds.). North-Holland.

Basket, F., R. Muntz, M. Chandy, and F. Palacios. 1975. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM* 22 (2):248–260.

Belady, Les A. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5 (2):78–101.

Bell, David, and Leonard LaPadula. 1976. Secure Computing System: Unified Exposition and Multics Interpretation. MITRE Technical Report 2547.

Bell, T., and M. Fellows. CSunplugged.org website. Video presentation of a workshop. http://csunplugged.org/videos.

Berners-Lee, Tim. 2000. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. Harper Business.

Birrell, Andrew, and Bruce Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems* 2 (1):39–59.

Boehm, Barry. 2002. Get ready for agile methods, with care. *IEEE Computer* (January):64–69.

Bohm, Corrado, and Giuseppe Jacopini. 1966. Flow diagrams, Turing machines, and languages with only two formation rules. *Communications of the ACM* 9 (5):366–371.

Brooks, Frederick. 1975. *The Mythical Man Month*. Addison-Wesley. Second edition published 1995.

Brooks, Frederick. 1986. No silver bullet—Essence and accident in software engineering. *Proceedings of the 10th IFIP World Congress*, pp. 1069–1076. Reprinted 1987 in *IEEE Computer* 20 (4):10–19.

Brooks, Frederick P. 1995. *The Mythical Man Month*, 2nd ed. Addison-Wesley. (Original work published 1975)

Brown, Doug, John Levine, and Tony Mason. 2012. *Lex & Yacc*, 2nd ed. O'Reilly Media. (Original work published 1992)

Brynjolfsson, Erik, and Andrew McAfee. 2012. *Race against the Machine: How the Digital Revolution Is Accelerating Innovation, Driving Productivity, and Irreversibly Transforming Employment and the Economy*. Digital Frontier Press.
Burke, James. 1985. *The Day the Universe Changed*. Little, Brown.

Burks, Arthur, Don Warren, and Jesse Wright. 1954. An analysis of a logical machine using parenthesis-free notation. *Mathematical Tables and Other Aids to Computation* 8:53–57.

Buzen, Jeffrey P. 1973. Computational algorithms for closed queueing networks. *Communications of the ACM* 16 (9):527–531.

Buzen, Jeffrey P. 1976. Fundamental operational laws of computer system performance. *Acta Informatica* 7:167–182.

Buzen, Jeffrey P. 2011. Computation, uncertainty, and risk. *ACM Ubiquity Symposium on Computation*. http://ubiquity.acm.org/article.cfm?id=1936886.

Carse, J. 1986. *Finite and Infinite Games*. Ballantine Books, Random House.

Cerf, Vinton, Yogen Dalal, and Carl Sunshine. 1974. Specification of Internet Transmission Control Program. RFC 675 (December). Available as https://tools.ietf.org/html/rfc675.

Cerf, Vinton, and Robert Kahn. 1974. A protocol for packet network interconnection. *IEEE Transactions on Communication Technology COM-22* (5):627–641.

Chaitin, Gregory. 2006. *Meta Math!: The Quest for Omega*. Vintage.

Chan, Tony, and Yousef Saad. 1986. Multigrid algorithms on the hypercube multiprocessor. *IEEE Transactions on Computers* C-35 (11):969–977.

Chaney, T. J., and C. E. Molnor. 1973. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers* 22 (April):421–422.

Cisco. 2012. Visual Networking Index (VNI) Forecast (2011–2016). http://cisco.com/en/US/netsol/ns827/networking_solutions_sub_solution.html.

Clark, David. 1988. The design philosophy of the DARPA Internet Protocol. *ACM Computer Communication Review* 18 (4):106–114.

Clark, Edmund. 2008. *The Birth of Model Checking. In 25 Years of Model Checking*, vol. 5000, ed. O. Grumberg and H. Veith, pp. 1–26. Lecture Notes in Computer Science. Springer Verlag.

Clark, Edmund, and E. Allen Emerson. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY*. Lecture Notes on Computer Science, Springer Verlag.

Clark, Richard. 2012. *Cyber War: The Next Threat to National Security and What to Do about It*. Ecco.

Codd, Edgar. 1970. A relational model of data for large shared databanks. *Communications of the ACM* 25 (2):109–117.

Codd, Edgar. 1990. *The Relational Model for Database Management*, 2nd ed. Addison-Wesley.

Coffman, Edward G. Jr., and Peter Denning. 1973. *Operating Systems Theory*. Prentice-Hall.

Comer, Douglas. 2013. *Internetworking with TCP/IP*, 6th ed., vol. 1. Addison-Wesley.

Cook, Stephen. 1971. The complexity of theorem proving procedures. *Proceedings of the 3rd ACM Symposium on Theory of Computation*, pp. 151–158. ACM Press.

Coplien, James, and Douglas Schmidt. 1995. *Pattern Languages of Program Design*. Addison-Wesley.

Coppersmith, Donald, and Shmuel Winograd. 1990. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation* 9:251–280.

Cormen, Thomas, Charles Leiserson, Ronald Rivest, and Clifford Stein. 2009. *Intro-*

duction to Algorithms. MIT Press.

Davies, Donald W., K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. 1967. A digital communication network for computers giving rapid response at remote terminals. *Proceedings of the ACM Symposium on Operating System Principles* (SOSP). http://doi.acm.org/10.1145/800001.811669

Dean, Jeffrey, and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. *Proceedings of the 6th Symposium on Operating System Design and Implementation 6* (OSDI'74). USENIX Association.

Dean, Jeffrey, and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51 (1):107–113.

Denning, Dorothy. 1976. A lattice model for secure information flow. *Communications of the ACM* 19 (5):236–243.

Denning, Dorothy. 1982. *Cryptography and Data Security*. Addison-Wesley.

Denning, Dorothy. 1998. *Information Warfare and Security*. Addison-Wesley.

Denning, Peter. 1968a. The working set model for program behavior. *Communications of the ACM* 11 (5):323–333.

Denning, Peter. 1968b. Thrashing: Its causes and prevention. *Proceedings of the AFIPS Conference* 32:915–922.

Denning, Peter. 1970. Virtual memory. *ACM Computing Surveys* 2 (3):153–189.

Denning, Peter. 1971. Third generation computer systems. *ACM Computing Surveys* 3 (4):175–216.

Denning, Peter. 1980. Working sets past and present. *IEEE Transactions on Software Engineering* SE-6 (1):64–84.

Denning, Peter. 1985. The arbitration problem. *American Scientist* 73 (Nov–Dec): 516–518.

Denning, Peter. 1987. Multigrids and hypercubes. *American Scientist* 75 (3):234–238.

Denning, Peter. 2001. Who are we? *Communications of the ACM* 44 (2): 15–19.

Denning, Peter. 2003. Great principles of computing. *Communications of the ACM* 46 (11):15–20.

Denning, Peter. 2007a. Computing is a natural science. *Communications of the ACM* 50 (7):15–18.

Denning, Peter. 2007b. The choice uncertainty principle. *Communications of the ACM* 50 (11):9–14.

Denning, Peter. 2013. Design thinking. *Communications of the ACM* 58:12.

Denning, Peter, Douglas Comer, David Gries, Michael Mulder, Allen Tucker, A. Joe Turner, and Paul Young. 1989. Computing as a discipline. *Communications of the ACM* 32 (1):9–23.

Denning, Peter, and Dennis Frailey. 2011. Who are we—now? *Communications of the ACM* 54 (6):27–29.

Denning, Peter, and Peter Freeman. 2009. Computing's paradigm. *Communications of the ACM* 52 (12):28–30.

Denning, Peter, Kevin Kahn, Jacques Leroudier, Dominique Potier, and Rajan Suri. 1976. Optimal multiprogramming. *Acta Informatica* 7 (2):197–216.

Denning, Peter, and Robert Kahn. 2010. The long quest for universal information access. *ACM Communications of the ACM* 53 (12):34–36.

Denning, Peter, and Craig Martell. 2004. Great Principles of Computing Web site. http://greatprinciples.org.

Denning, Peter, Walter Tichy, and James Hunt. 2000. Operating systems. In *Encyclopedia of Computer Science*, ed. A. Ralston, E. O'Reilly, and D. Hemmendinger, pp. 1290–1311. Nature Publishing Group, Grove's Dictionaries.

Denning, Peter J. 1991a. Queueing in networks of computers. *American Scientist* 79 (3):8–10.

Denning, Peter J. 1991b. In the queue: Mean values. *American Scientist* 79 (5):402–403.

Denning, Peter J., and Jeffrey P. Buzen. 1977. Operational analysis of queueing networks. *Modeling and Performance Evaluation of Computer Systems* (H. Beilner and E. Gelenbe, eds.), pp. 151–172. North-Holland.

Denning, Peter J., and Jeffrey P. Buzen. 1978. The operational analysis of queueing network models. *ACM Computing Surveys* 10 (3):225–261.

Denning, Peter J., and Tim Bell. 2012. The information paradox. *American Scientist* 100:470–477.

Dennis, Jack, and David Misunas. 1975. A preliminary architecture for a basic data-flow processor. *ACM 2nd Symposium on Computer Architecture*, pp. 126–132, ACM Press.

Dennis, Jack, and Earl Van Horn. 1966. Programming semantics for multiprogrammed computations. *Communications of the ACM* 9 (3):143–155.

Dennis, Jack, and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Communications of the ACM* 9 (3):143–155.

Després, Rémi. 2010. X.25 virtual circuits: Transpac in France—pre-Internet data networking. *IEEE Communications Magazine* 40 (11):40–46.

Dijkstra, Edsger. 1959. A note on two problems in connection with graphs. *Numerische Mathematik* 1:269–271.

Dijkstra, Edsger. 1965. Solution of a problem in concurrent program control. *Communications of the ACM* 8 (9):569.

Dijkstra, Edsger. 1968. The structure of the THE-multiprogramming system. *Communications of the ACM* 11 (5):341–346.

Dijkstra, Edsger. 1968a. Cooperating sequential processes. In *Programming Languages*,

ed. F. Genuys, pp. 43–112. Academic Press.

Dijkstra, Edsger. 1968b. The structure of the "THE" multiprogramming system. *ACM Communications* 11 (5):341–346.

Dretske, F. 1981. *Knowledge and the Flow of Information*. MIT Press.

Dreyfus, Hubert. 1972. *What Computers Can't Do*. MIT Press.

Dreyfus, Hubert. 1992. What Computers Still Can't Do. MIT Press. See also the Wikipedia summary at http://en.wikipedia.org/wiki/Hubert_Dreyfus%27s_views_on_artificial_intelligence.

Dreyfus, Hubert. 2001. *On the Internet*. Routledge.

Erlang, A. K. 1909. Theory of probabilities and telephone conversations. *Nyt Tidsskrift for Matematik B* 20.

Erlang, A. K. 1917. Solution of some problems in the theory of probabilities of significance in automatic telephone exchanges. *Electroteknikeren*, 13.

Fabry, Robert. 1974. Capability-based addressing. *Communications the ACM* 17 (7):403–412.

Flynn, Michael. 1972. Some computer organizations and their effectiveness. *IEEE Transactions on Computers* C-21:948–960.

Flyvbjerg, Bent, and Alexander Budzier. 2011. Why your IT project may be riskier than you think. *Harvard Business Review* (September):23–25.

Fortnow, Lance. 2009. The status of the P versus NP problem. *Communications of the ACM* 52 (9):78–86.

Fortnow, Lance. 2013. *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press.

Garey, Michael, and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Garfinkel, Simson. 2001. *Database nation: The Death of Privacy in the 21st Century*. O'Reilly.

Gelenbe, Erol. 2011. Natural computation. *ACM Ubiquity* (February). Available at http://ubiquity.acm.org/article.cfm?id=1940722.

Ginosar, R. 2003. Fourteen ways to fool your synchronizer. *Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, IEEE. Available at http://www.ee.technion.ac.il/~ran/papers/Sync_Errors_Feb03.pdf.

Gleick, J. 2011. *The Information: A History, a Theory, a Flood*. Random House.

Goldberg, Adele. 1983. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley.

Goldin, D., S. Smolka, and P. Wegner. 2010. *Interactive Computation: The New Paradigm*. Springer.

Goldstine, Herman. 1993. *The Computer from Pascal to von Neumann*. Princeton Uni-

versity Press.

Gordon, W. J., and G. F. Newell. 1967. Closed queueing systems with exponential servers. *Operations Research* 15 (2):254–265.

Graefe, Goetz. 2007. The five-minute rule twenty years later, and how flash memory changes the rules. *Proceedings of the 3rd International Workshop on Data Management on New Hardware*, article 6. ACM.

Gray, Jim, and Franco Putzolu. 1985. The 5 minute rule for trading memory for disk accesses. *Tandem Corporation Technical Report 86.1*. (Available at http://www.hpl.hp.com/techreports/tandem/TR-86.1.pdf)

Gries, David. 1971. *Compiler Construction for Digital Computers*. Wiley.

Gurevich, Yuri. 2012. What is an algorithm? In *Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science* (SOFSEM'12), Mária Bieliková, Gerhard Friedrich, Georg Gottlob, and Stefan Katzenbeisser, Eds., pp. 31–42. Springer-Verlag.

Habermann, A. Nico. 1969. Prevention of system deadlock. *Communications of the ACM* 12 (7):373–377.

Hafner, Katie, and Matthew Lyon. 1998. *Where Wizards Stay Up Late: The Origins of the Internet*. Simon & Schuster.

Brinch Hansen, Per. 1973. *Operating System Principles*. Prentice-Hall.

Harel, David. 1980. On folk theorems. *Communications of the ACM* 23 (7):379–389.

Hazen, Robert. 2007. *Genesis: The Scientific Quest for Life's Origins*. Joseph Henry Press.

Henderson, Harry. 2008. *Encyclopedia of Computer Science and Technology*. Facts on File.

Hennessey, John, and David Patterson. 2011. *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufman.

Hoare, C. A. R. 1974. Monitors: An operating system structuring concept. *Communications of the ACM* 17 (10):549–557.

Hoare, C. A. R. 1978. Communicating sequential processes. *Communications of the ACM* 21 (8):666–677.

Hoare, Tony. 1961. Algorithm 64: Quicksort. *Communications of the ACM* 4 (7):321.

Hoare, Tony. 1985. *Communicating Sequential Processes*. Prentice-Hall.

Hofstadter, Douglas. 1985. *Metamagical Themas: Questing for the Essence of Mind and Pattern*. Basic Books. See his essay "The genetic code: Arbitrary?"

Israel, Jay, James Mitchell, and Howard Sturgis. 1978. *Separating Data from Function in a Distributed File System*. Xerox Palo Alto Research Center Technical Report Series.

Jackson, J. R. 1957. Networks of waiting lines. *Operations Research* 5 (4):131–142.

Kahn, Robert, and Robert Wilensky. 1995. A framework for distributed digital object services. CNRI technical report, available as http://www.cnri.reston.va.us/cstr/arch/k-w.html. Reprinted 2006. *International Journal on Digital Libraries* 6 (2):115–123.

Kanerva, Pentti. 2003. *Sparse Distributed Memory*. MIT Press.

Karnaugh, Maurice. 1953. The map method for synthesis of combinational logic circuits. *Transactions of AIEE, Part 1,* 72 (9):593–599.

Karp, Richard. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, ed. R. E. Miller and J. W. Thatcher, pp. 85–103. Plenum.

Karp, Richard. 1993. Mapping the genome: Some combinatorial problems arising in molecular biology. *Proceedings of the 25th ACM Symposium on Theory of Computing,* ACM Press, 278–285.

Karp, Richard, and Raymond Miller. 1966. Properties of a model for parallel computations: Determinacy, termination, and queueing. *SIAM Journal of Applied Math* 14 (6):1390–1411.

Karp, Richard, and Raymond Miller. 1969. Parallel program schemata. *Journal of Computer and Systems Sciences* 3 (2):147–195.

Kilburn, T., D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. 1962. One level storage system. *IRE Transactions on Electronic Communication* (April):223–235.

Kinniment, D. J., and J. V. Woods. 1976. Synchronization and arbitration circuits in digital systems. *IEEE Proceedings* (October):961–966.

Kleene, Stephen. 1936. General recursive functions of natural numbers. *Mathematische Annalen* 112:727–742.

Kleinrock, Leonard. 1961. *Information flow in large communication nets.* RLE Quarterly Progress Report (July).

Kleinrock, Leonard. 1964. *Communication Nets: Stochastic Message Flow and Delay.* McGraw-Hill.

Kleinrock, Leonard. 1975. *Queueing Systems: Theory*, vol. 1. Wiley.

Kleinrock, Leonard. 1976. *Queueing Systems: Computer Applications*, vol. 2. Wiley.

Knuth, Donald. 1964. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM* 7 (12):735–736.

Kobayashi, H., and B. L. Mark. 2008. *System Modeling and Analysis*. Prentice-Hall.

Kurzweil, Ray. 2005. *The Singularity Is Near*. Viking.

Lamport, L. 1984. Buridan's principle. Technical report available from http://research.microsoft.com/users/lamport/pubs/buridan.pdf.

Lampson, Butler. 1974. Protection. *ACM SIGOPS Operating Systems Review* 8 (1):18–24.

Lampson, Butler. 1983. Hints for computer system design. *Proceedings of the ACM*

Symposium on Operating Systems Principles, pp. 33–48.

Lazowska, Edward D., John Zahorjan. G. Scott Graham, and Kenneth C. Sevcik. 1984.. *Quantitative System Performance*. Prentice-Hall.

Leiner, Barry, Vinton Cerf, David Clark, Robert Kahn, Leonard Kleinrock, Dan Lynch, Jon Postel, Larry Roberts, and Stephen Wolff. 1996. *Brief History of the Internet.* http://www.internetsociety.org/internet/what-internet/history-internet/brief-history-internet (retrieved March 2014).

Levy, Steven. 1984. *Capability-Based Computer Systems*. Digital Press.

Licklider, J. C. R. 1960. Man-computer symbiosis. *IRE Transactions on Human Factors in Electronics HFE-1* (March):4–11. Available at http://groups.csail.mit.edu/medg/people/psz/Licklider.html.

Licklider, J. C. R. 1963. *Memorandum for Members and Affiliates of the Intergallactic Computer Network*. DARPA memorandum (April). Available at http://worrydream.com/refs/Licklider-IntergalacticNetwork.pdf

Licklider, J. C. R., and Welden Clark. 1962. On-line man computer communication. *Proceedings of the AFIPS Conference SJCC* (May). Available from ACM at http://doi.acm.org/10.1145/1460833.1460847.

Liskov, Barbara, Alan Snyder, Russell Atkinson, and Craig Schaffert. 1977. Abstraction mechanisms in CLU. *Communications of the ACM* 20 (8):564–576.

Little, J. D. C. 1961. A proof for the queueing formula L = λW. *Operations Research* 9 (3):383–387.

Louden, Kenneth. 2011. *Programming Languages: Principles and Practice*, 3rd ed. Cengage Learning. (Original work published 2002)

Lukasiewicz, Jan. 1957. Aristotle's Syllogistic from the Standpoint of Modern Formal Logic. Oxford University Press.

MacCormick, John. 2012. *Nine Algorithms That Changed the Future*. Princeton University Press.

Madison, A. Wayne, and Alan P. Batson. 1976. Characteristics of program localities. *Communications of the ACM* 19 (5):285–294.

McMenamin, Adrian. 2011. *Applying Working Set Heuristics to the Linux Kernel.* Masters Thesis, Birkbeck College, University of London. Available at: http://cartesianproduct.files.wordpress.com/2011/12/main.pdf.

Menasce, Daniel A., and Virgilio A. F. Almeida. 2002. *Capacity Planning for Web Services*. Prentice-Hall.

Menasce, Daniel A., Virgilio A. F. Almeida, and Lawrence W. Dowdy. 1994. *Capacity Planning and Performance Modeling*. Prentice-Hall.

Metcalfe, Robert, and David Boggs. 1983. Ethernet: Distributed packet switching for local computer networks. *Communications of ACM* 26 (1):90–95.

Miller, Mark. 2003. Capability myths demolished. Johns Hopkins University, Systems Research Laboratory. Available at http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf.

Mitcham, Carl. 1994. *Thinking Through Technology: The Path Between Engineering and Philosophy*. University of Chicago Press.

Mockapetris, Paul. 1983. *Domain Names—Implementation and Specification.* Internet Engineering Task Force memoranda RFC 883 and 884 (November). Available at http://tools.ietf.org/html/rfc883, http://tools.ietf.org/html/rfc884

Moore, G. 1965. Cramming more components onto integrated circuits. *Electronics* 38 (8):4–6.

Negroponte, Nicholas. 1996. *Being Digital.* Vintage.

Nelson, Ted. 1980. *Literary Machines*. Mindful Press.

Neumann, Peter. 1995. *Computer-Related Risks*. ACM Press: Addison-Wesley.

Neumann, Peter G., Robert Boyer, Richard Feiertag, Karl Levitt, and Lawrence Robinson. 1980. A Provably Secure Operating System, its Applications, and Proofs. CSL-116 (2nd edition), SRI International, Menlo Park, CA (May).

Newell, Allen, Alan J. Perlis, and Herbert A. Simon. 1967. "Computer Science" [letter] *Science* 157 (3795):1373–1374.

Newell, Allen, and Herbert Simon. 1963. GPS: A program that simulates human thought. In *Computers and Thought*, ed. E. Feigenbaum and J. Feldman. McGraw-Hill.

Nilsson, Nils. 2010. *The Quest for Artificial Intelligence: A History of Ideas and Achievements*. Cambridge University Press.

Norman, Donald. 2010. Technology first, needs last: The research-product gulf. *Interactions (New York)* 17 (2):38–42.

Norman, Donald. 2013. *The Design of Everyday Things,* 2nd ed. Basic Books. (Previously published as *The Psychology of Everyday Things*, Basic, 1998)

Nyquist, Harry. 1928. Certain topics in telegraph transmission theory. *Transactions of AIEE* 47:617–644. Reprinted 2002 in *Proceedings of the IEEE* 90 (2).

Organick, Elliott. 1972. *The Multics System: An Examination of Its Structure.* MIT Press.

Organick, Elliott. 1973. *Computer System Organization: B5700–B6700 Series*. Academic Press.

Parnas, David. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15 (12):1053–1058.

Pierce, Benjamin. 2002. *Types and Programming Languages*. MIT Press.

Pouzin, Louis. 1973. Presentation and major design aspects of the CYCLADES computer network. *Proceedings of the NATO Advanced Study Institute on Computer Communication Networks,* pp. 415–434. Noordhoff International Publishing.

Pouzin, Louis. 1974. CIGALE, the packet switching machine of the CYCLADES computer network. *Proceedings of the IFIP Congress*, pp. 155–159.

Prieve, Barton, and Robert Fabry. 1976. VMIN—An optimal variable-space page replacement algorithm. *Communications of the ACM* 19 (5):295–297.

Pullen, J. Mark. 2000. *Understanding Network Protocols*. Wiley.

Quielle, J. P., and J. Sifakis. 1982. Specification and verification of concurrent systems in DESAR. In *Proceedings of the 5th International Symposium on Programming*, 337–250.

Ralston, Anthony, Edwin Reilly, and David Hemmengdinger. 2003. *Encyclopedia of Computer Science*. 4th ed., Wiley.

Reiser, Martin, and Stephen Lavenberg. 1980. Mean value analysis of closed multichain queueing networks. *Journal of the ACM* 27:313–322.

Rice, John. 1976. Algorithmic progress in solving partial differential equations. CS Technical Report CSD-TR-173 (Jan). Available at http://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1117&context=cstech.

Ritchie, Dennis, and Kenneth Thompson. 1974. The UNIX Time-Sharing System. *Communications of the ACM* 17 (7):365–375.

Rivest, Ronald, Adi Shamir, and Len Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21 (2):120–126.

Roberts, Larry. 1967. Multiple computer networks and intercomputer communication. *Proceedings of the ACM Symposium on Operating Systems Principles* (Gatlinburg, October).

Roberts, Larry, and T. Merrill. 1966. Toward a cooperative network of time-shared computers. *AFIPS Conference Proceedings FJCC* (October).

Rocchi, Paolo. 2010. *Logic of Analog and Digital Machines*. Nova Publishers.

Rocchi, P. 2012. *The Logic of Digital and Analog Machines*. Nova Publishers.

Rosenbloom, Paul S. (November 2004). A new framework for computer science and engineering. *IEEE Computer* 31–36.

Rosenbloom, Paul S. 2012. *On Computing: The Fourth Great Domain of Science*. MIT Press.

Russell, Stuart, and Peter Norvig. 2010. *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall.

Saltzer, Jerome. 1965. *Traffic Control in a Multiplexed Computer System*. MIT Project MAC-TR-30 (Sc.D. thesis). Available at http://web.mit.edu/Saltzer/www/publications/MIT-MAC-TR-030.ocr.pdf.

Saltzer, Jerry, and Frans Kaashoek. 2009. *Principles of Computer System Design*. Morgan-Kaufman.

Saltzer, Jerry, David Reed, and David Clark. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems* 2 (4):277–288.

Saltzer, Jerome, and Michael Schroeder. 1975. Protection of information computer systems. *Proceedings of the IEEE* 63 (9):1278–1308.

Sayre, David. 1969. Is automatic "folding" of programs efficient enough to displace manual? *Communications of the ACM* 12 (12):656–660.

Scheifler, Robert, Ron Newman, and James Gettys. 1988. *X Window System: C Library and Protocol Reference*. Digital Press.

Scherr, Allan. 1965. *An Analysis of Time-Shared Computer Systems*. MIT Technical report MIT-LCS-TR-018. Available at http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-018.pdf.

Schneier, Bruce. 2004. *Secrets and Lies: Digital Security in a Networked World*. Wiley.

Schneier, Bruce. 2008. *Schneier on Security*. Wiley.

Searle, John. 1984. *Minds, Brains and Science*. Harvard University Press.

Seitz, C. L. 1980. System timing. In *Introduction to VLSI Systems*, ed. C. Mead and L. Conway, 218–262. Addison-Wesley.

Shannon, Claude. 1948. The mathematical theory of communication. *Bell System Technical Journal* 27:379–423, 623–656.

Shannon, Claude, and Warren Weaver. 1949. *The Mathematical Theory of Communication*. University of Illinois Press (reprinted 1998). Shannon's original paper is available on the Web: http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html

Shor, Peter. 1994. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. ACM Press. Reprinted in *SIAM J. Computing* 1997:1484–1509.

Simon, Herbert. 1996. *The Sciences of the Artificial*, 3rd ed. MIT Press.

Spector, Alfred. 1982. Performing remote operations efficiently in a local computer network. *Communications of the ACM* 25 (April):246–260.

Standage, Tom. 2003. *The Turk: The Life and Times of the Famous Eighteenth-Century Chess-Playing Machine*. Berkeley Trade.

Stewart, William J. 2009. *Probability, Markov Chains, and Simulation*. Princeton University Press.

Strassen, Volker. 1969. Gaussian elimination is not optimal. *Numerische Mathematik* 13:354–356.

Sutherland, Ivan. 2012. The tyranny of the clock. *Communications of the ACM* 55 (1) (October): 35–36.

Sutherland, I., and J. Ebergen. 2002. Computers without clocks. *Scientific American*

(August), 62–69.

Tanenbaum, Andrew. 1980. *Computer Networks* (5th ed. with David Wetherall published 2010). Prentice-Hall.

Tanenbaum, Andrew, and Sape Mullender. 1981. An overview of the Amoeba distributed operating system. *ACM SIGOPS Operating Systems Review* 15 (3):51–64.

Tedre, Matti. 2014. *In Search of the Science of Computing*. Taylor & Francis.

Turing, Alan. 1937. On computable numbers with an application to the Entscheidungs problem. *Proceedings of the London Mathematical Society* 2:230–265.

Turing, Alan. 1950. Computing machinery and intelligence. *Mind* 59 (236): 433–460.

von Ahn, L., and L. Dabbish. 2004. Labeling images with a computer game. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pp. 319–326. ACM Press.

von Neumann, John. 1945. *First Draft of a Report on the EDVAC*. In *IEEE Annals of the History of Computing* 15 (4):27–43. (Original work published as a technical report from the US Army project at the University of Pennsylvania, 1945)

Wikipedia. 2014. List of programming languages. Available http://en.wikipedia.org/wiki/List_of_programming_languages

Wilkes, Maurice. 1968a. *Time Sharing Computer Systems*. Elsevier North-Holland.

Wilkes, Maurice. 1968b. "Computers then and now." 1967 Turing Lecture. *Journal of the ACM* 15 (1):1–7.

Wilkes, Maurice. 1985. *Memoirs of a Computer Pioneer*. MIT Press.

Wilkes, Maurice. 1995. *Computing Perspectives*. Morgan Kaufman.

Wilkes, Maurice, and Roger Needham. 1979. *The Cambridge CAP Computer and Its Operating System*. Elsevier North-Holland.

Willinger, Walter, David Alderson, and John Doyle. 2009. Mathematics and the Internet: A source of enormous confusion and great potential. *Notices of the AMS* 56 (5):586–599.

Winograd, Terry. 1996. *Bringing Design to Software*. ACM Press Addison-Wesley.

Winograd, Terry, and Fernando Flores. 1987. *Understanding Computers and Cognition: A New Foundation for Design*. Addison-Wesley.

Wirth, Niklaus. 1989. *Programming in Modula 2*. Springer-Verlag.

Wolfram, Stephen. 2002. *A New Kind of Science*. Wolfram Media.

Wulf, William, W. Corwin, Anita Jones, Roy Levin, C. Pierson, and F. Pollack. 1974. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM* 17 (6):337–345.

Xiang, Xiaoya, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: A higher order theory of locality. *ACM Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS'13), pp. 343–356.

Zimmerman, Hubert. 1980. OSI reference model—The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications COM-28* (4):425–432.

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

C

D

E

F

G

H

N

O

P

T

伟大的计算原理

Great Principles of Computing

理解计算，并不是将计算机当作工具，而是要领会应用于计算诸多方面和示例中的基础原理。本书为我们铺设了一条发掘这些基础原理的道路。一旦读完了Denning和Martell为我们撰写的这本精心之作，你会发现这本书不是填鸭式地灌输，而是需要你自己去发掘、吸收，并将它们连贯起来。本书全力为我们展现这样一个全面的视图：计算是什么以及计算又是如何应用到我们居住的这个世界的。

—— Leonard Kleinrock，加州大学洛杉矶分校计算机科学特聘教授

通过努力，几乎每个人都可以学会编程。然而仅仅靠编写代码并不足以构建辉煌的计算世界，要完成这个目标起码需要对以下几个方面有更深入的认识：计算机是如何工作的，如何选择算法，计算系统是如何组织的，以及如何进行正确而可靠的设计。如何开始学习这些相关的知识呢？本书就是一个方法——这是一本深思熟虑地综合描述计算背后的基本概念的书。通过一系列详细且容易理解的话题，本书为正在学习如何认识计算（而不仅仅是用某种程序设计语言进行编码）的读者提供了坚实的基础。Denning和Martell确实为计算领域的学生呈现了其中的基本原理。

—— Eugene H. Spafford，普渡大学计算机科学教授

计算通常被看作是一个按照摩尔定律高速发展的技术领域。如果我们稍不留意，就有可能错过一个划时代的技术突破或者翻天覆地的理论发展。该书从一个不同的视角，把计算看作一门遵从一些基本原理的科学，而这些基本原理可以涵盖其中所有的技术。计算机科学是一门关于信息处理的科学，我们需要一种新的“语言”来描述这门科学。在本书中，Denning和Martell给出了一个重要原理框架作为这种语言。本书涵盖了计算的方方面面——包括算法、体系结构和设计。

Denning和Martell将计算的基本原理分为六大类：通信、计算、协作、记忆（存储）、评估和设计。他们首先对作为科学的计算进行了概括性的描述，包括它的历史、与其他领域的诸多交互、应用领域以及重要原理框架的结构。进而在不同的领域（包括信息、机器、程序设计、计算、存储、并行、排队和设计）中验证了这些基本原理。最后，将这些基本原理应用到计算机网络，尤其是因特网之中。

无论是包含计算分支的科学和工程领域中的专业人员，还是想要大致了解计算机科学中不太熟悉的计算领域的从业人员，抑或是想要窥视计算机科学门径的非计算机科学专业人员，本书都是适合他们阅读的一本基础读物。

作者简介

彼得 J. 丹宁（Peter J. Denning） 美国海军研究生院杰出教授，ACM前主席，著名计算机杂志《Communications of The ACM》前主编。

克雷格 H. 马特尔（Craig H. Martell） 美国海军研究生院计算机科学系副教授。

上架指导：计算机科学

ISBN 978-7-111-5

9 787111 567264 >

定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

[General Information]
书名=伟大的计算原理
作者=（美）彼得J.丹宁（Peter J.Denning），（美）克雷格H.马特尔（Craig H.Martell）著
丛书名=计算机科学丛书
页数=239
SS号=14200223
出版日期=2017.05
出版社=北京：机械工业出版社
ISBN号=978-7-111-56726-4
中图法分类号=TP3
原书定价=69.00
主题词=计算机科学
参考文献格式=（美）彼得J.丹宁（Peter J.Denning），（美）克雷格H.马特尔（Craig H.Martell）著.伟大的计算原理[M].北京：机械工业出版社,2017.05.